MASTER EN SCIENCES INFORMATIQUES

Evaluation de règles SWRL en chaînage avant

Busard, François

Award date:
2008

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année Académique 2007 - 2008

Evaluation de règles *SWRL* en chaînage avant

François BUSARD
Juin 2008

Mémoire présenté en vue de l'option du grade de Maître en informatique

Résumé

A l'heure actuelle, le *World Wide Web* regroupe de plus en plus d'informations. Celles-ci sont traitées automatiquement par des agents intelligents afin de rendre possible sa communication et ses échanges avec d'autres applications. Cependant l'information décrite sur la Toile est structurée dans le but d'être consultée par les humains et non pas par des agents automatiques. Le consortium du *World Wide Web* s'est penché sur ce problème et a proposé, il y a quelques années, les langages *OWL* et *SWRL* facilitant le traitement automatique de l'information. Ces langages font parties des technologies du *Web sémantique* et augmentent l'expressivité du *Web* en ajoutant la sémantique des données.

Dans ce mémoire, le langage de règles *SWRL* est étudié afin de comprendre comment il permet le traitement des connaissances. L'évaluation des règles *SWRL* permet la découverte de nouvelles informations implicitement exprimées. Cette évaluation peut être effectuée soit en chaînage avant, soit en chaînage arrière. L'objectif de ce mémoire est d'analyser l'évaluation en chaînage avant et de montrer comment l'appliquer au langage de règles *SWRL*. Ensuite, une méthode d'évaluation en chaînage avant de ces règles est présentée. Pour terminer, les différents procédés publiés dans la littérature informatique sont décrits et servent de base pour critiquer la méthode proposée.

Abstract

The current *World Wide Web* aggregates more and more information. These are treated automatically by intelligent agents in order to be communicated to and to be shared with other applications. However, the information described on the web is structured to be seen by humans. Therefore the intelligents agents can't understand it. To tackle this problem, the *World Wide Consortium* proposed the *OWL* and *SWRL* languages. These languages faciliate the automatic treatment of information and they are a part of the *Semantic Web* technologies. The *OWL* and *SWRL* languages add semantic information to the data.

In this thesis, the *SWRL* rule language is studied in order to understand how it allows to treat knowledge. Thanks to the evaluation of the *SWRL* rules, new facts are discovered. This evaluation can be done either by backward chaining or by forward chaining. The goal of this thesis is to analyse the evaluation by forward chaining and to show how it can be applied to the *SWRL* rule language. Later, a methodology to evaluate rules by forward chaining is explained. At the end of this thesis, different processes found in the computer science literature are described and they serve of basis to criticise the methodology proposed.

keywords : *OWL, SWRL, forward chaining, Magic sets transformation, Rete algorithm*

Remerciements

Au terme de cet ouvrage, je tiens à remercier tous ceux sans qui ce travail n'aurait pu être mené à bien.

Je tiens d'abord à exprimer ma gratitude envers le promoteur de ce mémoire, Wim Vanhoof pour le suivi et les nombreux conseils dont il m'a gratifiés.

J'adresse une pensée toute particulière à Michel Vanden Bossche afin de le remercier de l'accueil qu'il m'a réservé au sein de son entreprise *Mission Critical*. Il ne m'a pas seulement ouvert les portes de son entreprise, il m'a fait découvrir une vision de l'informatique hors du commun. Je remercie aussi Vincianne Capelle qui a veillé à mon intégration et à mon bien-être tout au long de mon séjour dans l'entreprise.

Je tiens à remercier Ludovic Lagevine dont les idées lumineuses et les connaissances scientifiques ont guidé mon travail tout au long de cette année ainsi que Nikolay Pelov pour ses nombreuses explications concernant les technologies utilisées à l'entreprise. Je remercie également le reste de l'équipe de *Mission Critical* pour leur accueil et leurs conseils avisés.

La réalisation de ce mémoire n'aurait pas été possible sans le soutient moral de ma famille. Un merci tout particulier à mon frère Simon et à mon papa pour leurs nombreuses relectures de ce mémoire et pour les conseils dont ils m'ont faits part.

Merci à Hélène Gennart de m'avoir soutenu pendant mes périodes de doutes, et pour ses encouragements répétés.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

La définition du cadre de ce travail

Le point de départ de ce mémoire est le travail réalisé au cours d'un stage à l'entreprise *Mission Critical*. Cette entreprise développe, depuis une dizaine d'années, des outils informatiques indispensables à la réalisation de l'activité de ses clients.

La première partie de ce chapître présente l'entreprise *Mission Critical* avant de décrire les objectifs et la structure de ce mémoire.

1.1 L'entreprise *Mission Critical*

Les idées à la base de la rédaction de cette partie du mémoire sont tirées de l'article [Bos01].

L'entreprise *Misson Critical* crée des solutions informatiques de bonne qualité en tenant compte des difficultés dues à la complexité et à la diversité des domaines à modéliser ainsi qu'à la pression exercée par les changements. Cette entreprise évolue dans un monde informatique en pleine crise. En effet, les études effectuées dans ce domaine montrent que 80% des coûts d'un projet sont destinés à la maintenance. Afin de pallier ce problème, il existe différentes approches. Il est possible de modéliser le domaine du client au moyen de méthodes formelles telles que *VDM* [Jon90], *SDL* [GGP03], *Z* [ASM80] ou *B* [Sch01]. Ces méthodes sont coûteuses car il faut combler l'écart entre les formalismes mathématiques et leur implémentation basée sur des langages traditionnels de programmation impérative. Une autre approche, utilisée par l'entreprise *Mission Critical*, consiste à recourir aux langages déclaratifs tels que *ML* [MTH90], *Haskell* [Bir98], *Mercury* [HCS+08], etc. Ces langages sont basés sur les mathématiques pures. *Mission Critical* reste une des rares entreprises au monde à utiliser de tels langages afin d'implémenter des programmes informatiques dans les entreprises.

La présentation des critères se rapportant aux langages déclaratifs reprise ci-dessous, aide à comprendre comment ces langages peuvent répondre à la crise du logiciel.

Le critère de conformité : La différence entre la spécification du domaine et son implémentation dans un langage déclaratif est minime ;

Le critère de performance : De bonnes performances sont atteintes en créant d'abord un programme correct (conforme) et en le transformant ensuite en un programme performant. Cette transformation est nécessaire car il est très difficile de produire un programme conforme et performant à la fois ;

Le critère de réutilisation : Ces langages réutilisent des morceaux de programmes (librairies, structures de données, etc.) facilement paramétrisés ;

Le critère d'adaptabilité : Ces langages supportent les changements ;

Le critère de mise à l'échelle : Ces langages permettent la mise à l'échelle sans augmenter les coûts du système.

Pour les raisons citées ci-dessus, à l'entreprise *Mission Critical*, les solutions informatiques sont développées dans le langage déclaratif *Mercury*.

1.1.1 Les technologies et les projets de l'entreprise

Malgré l'utilisation d'un langage déclaratif, l'entreprise *Mission Critical* s'est rendue compte que certains problèmes subsistaient lors du développement de projets informatiques. L'entreprise a identifié que ces problèmes étaient liés au fait qu'il existe un écart trop important entre les besoins exprimés par les utilisateurs et la solution fournie au client. Le client possède des connaissances reprises dans différentes structures de données regroupant tous les savoirs de son entreprise. Bien souvent, ces connaissances sont informelles. Cependant, les technologies de traitement de l'information nécessitent des connaissances formalisées. Afin de pallier ce problème, l'entreprise *Mission Critical* utilise les ontologies pour décrire formellement les domaines de connaissances. L'avantage de ces descriptions réside dans le fait qu'elles sont compréhensibles par les humains (et même par les non informaticiens) et par les ordinateurs. Dès lors, le client peut contribuer à la modélisation de son domaine de connaissances. Une fois cette opération réalisée, la description est communiquée aux experts en informatique afin qu'ils l'analysent et créent le logiciel répondant aux besoins du client. Il faut préciser que les descriptions des connaissances sont indépendantes du projet informatique considéré. De ce fait, l'entreprise *Mission Critical* a mis sur pied une plateforme appelée *ODASE (Ontology Driven Architecture for Software Engineering)* capable de générer automatiquement le code d'une description donnée. Lors de l'exécution de ce code, de nouveaux faits peuvent être découverts en utilisant les outils de raisonnement.

Les ontologies fournissent les concepts pour modéliser un domaine de connaissances. En pratique, elles sont exprimées à l'entreprise *Mission Critical* au moyen du langage *OWL*. Le langage des règles *SWRL* est utilisé pour augmenter l'expressivité limitée du langage *OWL*. Ces langages, proposés par le *Web sémantique*, seront décrits dans la suite de ce mémoire. Les langages *OWL* et *SWRL* sont des sous-langages de la logique de premier ordre tout comme le langage *Mercury* qui implémente la plateforme *ODASE*. L'appartenance de ces langages à un sous ensemble de la logique du premier ordre facilite le traitement des descriptions.

1.2 L'objectif de ce mémoire

La plateforme *ODASE* est capable de découvrir de nouvelles connaissances. Pour ce faire, cette plateforme dispose d'un raisonneur conçu pour évaluer les règles et inférer en chaînage arrière de nouvelles informations concernant le domaine modélisé. Toutefois, il existe une autre technique d'évaluation des règles appelé chaînage avant. L'objectif de ce mémoire est de proposer une méthode d'évaluation en chaînage avant d'un ensemble de règles *SWRL*. Dans ce mémoire, le travail du raisonneur sur le langage de règles *SRWL* est analysé avant de développer un algorithme permettant d'évaluer efficacement un ensemble de règles en chaînage avant.

Dans le chapître 2, les technologies du *Web sémantique* utiles à la modélisation d'un domaine seront présentées de manière précise. Le chapître 3 décrira le processus de raisonnement qu'il est possible d'appliquer aux règles *SWRL* ainsi que les principes guidant l'évaluation des règles en chaînage avant. A la fin de cette partie, deux méthodes d'optimisation du processus de chaînage avant seront exposées. Le chapître 4 présentera la méthodologie mise au point ayant servi de base à l'implémentation de l'engin développé durant le stage à l'entreprise *Mission Critical*. A la fin de ce mémoire, dans le chapître 5, le bilan ainsi que les perspectives de ce travail seront détaillés.

Chapitre 2

Une nouvelle vision du *W.W.W*

Depuis la nuit des temps, les êtres humains tentent de se comprendre par l'intermédiaire de divers langages. Au début de leur histoire, ils communiquaient par des gestes et des cris puis au fil du temps, ils ont mis au point des procédés structurés afin de faciliter l'échange d'information.

A l'époque actuelle, grâce au développement de ces langages, les êtres humains sont capables d'exprimer et de transmettre la plus grande partie de leurs idées. La grande difficulté à laquelle doivent faire face les personnes qui communiquent réside dans le fait que l'échange d'information est sujet à l'interprétation. Le récepteur doit saisir le sens initié par la personne qui a formulé l'information sous peine de mal comprendre les idées transmises.

Ce problème est probablement dû au fait que la plupart des mots ont plusieurs significations de telle sorte que la compréhension d'un groupe de mots repose très souvent sur l'interprétation du contexte. Afin de pallier à cette difficulté de communication, les Hommes ont très souvent recours à des schémas plus ou moins formels. Grâce à ceux-ci, l'interprétation est fortement limitée car la signification des composants est admise préalablement par les intervenants de l'échange.

Il n'est donc pas facile pour les êtres humains de se comprendre et il en est de même pour les ordinateurs qui, contrairement à eux, ne sont pas doués d'intelligence et ne peuvent interpréter l'information. L'ordinateur doit donc lui aussi, avoir recours à des représentations formelles admises par tous afin de faciliter la compréhension de l'information qu'il échange sur le *Web* avec les autres machines.

Dans son état actuel, le *World Wide Web* ne permet pas l'échange automatique de l'information. Pour comprendre pourquoi il en est ainsi, il faut revenir à ses débuts.

En 1989, Tim Berners Lee, invente le *World Wide Web* en créant le premier serveur Web et un programme client capable de se connecter à ce serveur. En 1990, il propose le *Hypertext Markup Language (HTML)*, un langage décrivant le format de documents web contenant des liens hypertextes. *HTML* décrit comment l'information doit être affichée à l'utilisateur qui consulte une page *Web*. Ce format devient le langage le plus utilisé sur la

Toile mais reste centré sur l'utilisateur. En effet, il ne permet pas la transmission de la signification des éléments mais seulement de la mise en page des documents [Jac08].

En 1998, apparait *Extensible Markup Language (XML)* qui permet l'échange de données entre des machines distantes. Les avantages de ce langage sont sa simplicité et qu'il permet de donner une structure aux données dont le format est très flexible. Par contre, il ne dit rien à propos de la signification des données et il n'y a pas de convention pour décrire les éléments. Pour les même concepts à modéliser, deux utilisateurs peuvent fournir des fichiers *XML* complètement différents. Avant de pouvoir interpréter un fichier de ce langage, il faut en connaitre sa structure. Dès lors, il est impossible d'automatiser l'analyse de ce fichier car sa structure est propre à la personne qui l'a écrit [Jac08].

Il est aisé de se rendre compte que les langages *HTML* et *XML* comme ils ont été définis à la base, ne fournissent pas le moyen automatique d'interpréter les éléments présents sur le *Web*.

Comme il est décrit dans l'article de Nigel Shadbolt, Wendy Hall et de Tim Berners-Lee [SHBL06] sur le *Web sémantique*, l'objectif de la nouvelle vision est de corriger les défauts du *Web* actuel trop centré sur l'utilisateur en augmentant son expressivité afin d'améliorer l'échange des données. Pour le moment, les programmes qui s'échangent de l'information sur la Toile sont des agents et des robots intelligents qui exécutent des tâches automatiques. Ceux-ci sont incapables d'opérer sur des données hétérogènes car ils sont dédiés à un type de tâche particulier dont ils connaissent la structure des données à traiter. L'amélioration du *Web* doit augmenter l'interopérabilité des systèmes agents et des robots intelligents afin qu'ils soient capables de comprendre les tâches qui leur sont assignées.

Pour atteindre cet objectif, des nouveaux standards sont définis dans le but de décrire les données de manière uniforme. Dès lors, il est possible de développer des agents automatiques capables de travailler avec n'importe quel objet issus du *W.W.W.*. De plus, il est possible de partager, d'intégrer et de dériver des données issues de différentes sources créées par des communautés différentes [Jac08].

Le *World Wide Web Consortium (W3C)* et d'autres partenaires ont développé des technologies regroupées sous le nom de *Web sémantique* définissant des structures de données communes en vue d'échanger, de réutiliser ces données au sein de différentes applications. Ces technologies visent à obtenir un *Web* orienté vers les données afin que n'importe quelles applications puissent traiter n'importe quelles données. Le *Web sémantique* se base sur le Web actuel auquel il augmente la formalisation afin d'obtenir un nouveau *Web* [Her07].

Le *Web sémantique* regroupe un bon nombre de technologies mais seulement le *Resource Description Framework*, l'*Ontology Web Language* et le *Semantic Web Rule Language* seront abordés dans le cadre de ce travail. En outre, les idées du Consortium sont plus ambitieuses que de simplement échanger des données sur le *World Wide Web* car le *Semantic Web Rule Language* consiste à adapter une partie de la logique mathématique au nouveau *Web* [Her07].

Pour atteindre son objectif, le *Web sémantique* propose un modèle en couches où chaque couche représente une technologie qui apporte un degré d'expressivité supérieur à la technologie de la couche inférieure. Ce modèle est représenté par le schéma 2.1, page 13 .

Dans ce modèle, la couche reprenant *Unicode* et *Uniform Resource Identifier (URI)* garantit que les caractères utilisés par ces technologies sont bien les caractères internationaux. En outre, une *URI* est attribuée à chaque objet afin de l'identifier sur le réseau. Les informations sont décrites avec le langage *XML*, des *Name Space* et des *Schémas XML* afin d'intégrer le *Web sémantique* avec les autres standards *XML* déjà en vigueur. *RDF* et ses schémas décrivent les assertions concernant les objets présents sur le *Web*. Cette technologie définit un vocabulaire commun pour représenter les éléments du monde réel. La couche *Ontolog vocabulary* permet d'exprimer les relations entre les différents concepts. Vient ensuite la couche *Logic* permettant d'écrire des règles concernant les concepts. Ces deux dernières couches sont exprimées de manière théorique et ce sont respectivement *Ontology Web Language* et *Semantic Web Rule Language* qui implémentent ces principes. La couche *Proof* exécute les règles et les évalue en coopération avec la couche *Trust*. Celle-ci identifie les règles correctes et celles qui ne le sont pas. Certaines couches possèdent un mécanisme de signature digitale (*Digital Signature*) détectant les altérations des documents échangés [KM01].

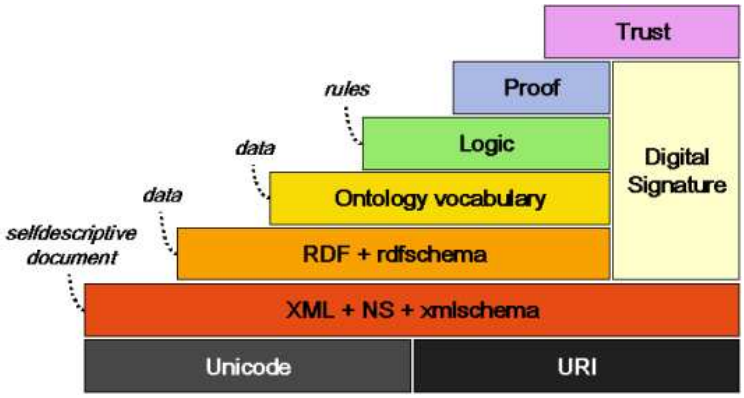

FIG. 2.1 – Le modèle en couches proposé par le *W3C*

Dans la suite de ce rapport, certaines technologies du schéma ci-dessus vont être détaillées afin de montrer comment elles contribuent à l'échange automatique de données entre les différents agents du web.

2.1 Resource Description Framework (*RDF*)

Cette partie du travail présente le langage *RDF* et montre comment celui-ci intervient dans le modèle en couches (voir figure 2.1 page 13).

Comme il a été vu précédemment, *RDF* offre la possibilité de décrire les entités (aussi appelées ressources) du monde réel ainsi que d'effectuer des assertions sur ces dernières. Ce langage exprime un ensemble de faits propres à un domaine particulier. Cependant, il ne s'échange directement entre les machines car cette technologie ne s'intègre pas au *Web* actuel. Les faits *RDF* sont donc transformés en une structure *XML* commune. La mise sur pieds d'une telle structure de données permet de développer des analyseurs syntaxiques et des outils indépendants d'une technologie et d'un organisme de développement. Grâce à eux, l'information contenue dans la structure peut être utilisée dans un cadre différent pour lequel elle a été définie. Cette vision offre de nouvelles possibilités d'exploitation des éléments décrits sur le *Web* et rend l'accès aisé aux données à n'importe quel utilisateur.

Cette partie de ce mémoire justifie le recours au langage *RDF* en montrant les limites du langage *XML*. Ensuite, les spécificités du *Resource Description Framework* sont présentées et pour terminer, ses limites sont exposées. L'ensemble des idées reprises dans cette partie sont très largement inspirées du document du *W3C* [MM04].

2.1.1 Les limitations de *XML*

La recommandation du *W3C* concernant le langage *XML* exprimée fin des années 90 ne permet pas l'échange automatique des ressources sur le *Web*. En effet, ce métalangage donne une représentation des connaissances et non un format commun des messages. La représentation est propre à chaque application qui échange ces fichiers sur *Internet*. Avant de créer un fichier *XML* en vue d'être lu par des agents web, une définition de type de document (*Document Type Definition (DTD)*) doit être spécifiée afin de donner un sens aux balises présentes dans le fichier. Lors de l'élaboration d'un agent web, si le développeur a pris en compte cette définition, alors le programme sera capable d'interpréter les balises et de ce fait, il pourra comprendre la signification des données. Dans le cas contraire, *XML* ne fournissant pas la sémantique des éléments qu'il décrit, l'agent web sera incapable de traiter le fichier. L'échange automatique des données sur le web n'est possible que si toutes les structures de données de chaque fichier *XML* créé respectent un formalisme.

Le langage *RDF* permet une plus grande généricité que le méta-langage *XML* en proposant une structure commune de descriptions des données. De ce fait, il n'est plus nécessaire de connaître au préalable la structure spécifique des fichiers car celle-ci est la même pour tous appartenant à ce type.

2.1.2 Description du langage *RDF*

Présentation

Le langage *RDF* est un modèle de données décrivant des objets, des entités et leurs relations. Il fournit le moyen d'ajouter des faits et des données définissant l'état de ces entités par le biais d'assertions. Par exemple, le fait qu'Alice a un enfant dont le prénom est Bob peut être modélisé en *RDF*.

Les différents états des entités peuvent être représentés par plusieurs formes. La plus visuelle est un graphe composé d'arcs et de noeuds mais les états peuvent aussi être vus sous la forme de triples *RDF* composés d'un *Sujet*, d'un *Prédicat* et d'un *Objet*. Et pour terminer, la sérialisation du fichier *RDF* se fait via une extension du langage *XML* appelée *RDF/XML* qui permet l'échange des concepts à travers le *Web*. Ces différentes formes de représentation seront détaillées dans les parties suivantes.

La description des ressources

La description des ressources se fait au moyen d'assertions. Chacune d'elles décrit une chose, une propriété de l'objet sur lequel elle porte ou définit une valeur de propriété pour cet objet. Selon le formalisme, dans toutes les assertions apparaissent un *Sujet*, un *Prédicat* et un *Objet*. Le *Sujet* est la partie qui identifie l'objet sur lequel porte l'assertion, le *Prédicat* identifie la propriété portant sur le *Sujet*. Et pour finir, l'*Objet* est la valeur de la propriété pour ce *Sujet*.

Dans l'assertion citée en exemple précédemment, des noms ont été utilisés pour référencer les ressources. Comment distinguer deux *Sujet* ayant le même nom mais mais définissant des ressources différentes ?

Pour cela, il est nécessaire d'avoir recours à un identifiant pour différencier chaque *Sujet*, chaque *Prédicat* et chaque *Objet* quand cela est possible. Le *Web* dispose déjà d'un tel système avec les *Uniform Resource Locator (URL)* qui sont constituées de chaînes de caractères. Celles-ci identifient une page *Web* mais constituent aussi le mécanisme d'accès à cette page. Cette dernière particularité n'étant pas nécessaire au langage *RDF*, une *Uniform Resource Identifier (URI)* est associée à chacune des ressources du modèle afin de l'identifier. Les *URL* sont un cas particulier des *URI* par le fait que ces dernières ne constituent pas impérativement son mécanisme d'accès. Pourtant l'utilisation d'*URI* peut sembler renvoyer à l'adresse d'un site web mais il n'en est rien. Dans le langage *RDF*, son rôle est simplement de distinguer un *Sujet*, un *Prédicat* ou un *Objet* parmi d'autres indépendamment du fait que la ressource décrite se trouve sur le *W.W.W.*. L'utilisation de ce mécanisme constitue le moyen d'éviter toute ambiguïté de sens intrinsèque aux mots employés. Chaque *URI* correspond à un terme particulier pour lequel la sémantique est acceptée par ses utilisateurs. De cette façon toutes les incertitudes dues au langage sont levées.

Dans le langage *RDF*, une *URI référence (URIref)*, constituée d'une *URI* et d'un fragment optionnel à la fin, est utilisée pour distinguer les éléments. Le *Prédicat* d'une

assertion est identifié par une *URIref* tandis que ce n'est pas nécessairement vrai pour le *Sujet* et l'*Objet*. Il est bon de noter qu'il est conseillé d'utiliser un maximum d'*URIref* à la place des mots dans une assertion. N'importe qui peut fixer une *URIref* faisant référence à des termes déjà définis mais dans ce cas, l'harmonisation des descriptions des ressources ne peut-être atteinte. Faire appel au vocabulaire déjà définit, permet au programme traitant ces données de faire des recoupements entre les différents concepts et déduire de nouvelles informations.

Le modèle *RDF*

Il a été spécifié dans une partie antérieure de ce travail que des faits *RDF* pouvaient être vus comme un graphe de noeuds et d'arcs. Pour l'assertion d'un fait, un premier noeud est créé contenant le *Sujet* et un second contenant l'*Objet*. Le premier noeud est relié au second par un arc dont l'étiquette est le *Prédicat* de l'assertion. Lorsque le *Sujet* et l'*Objet* ne sont pas définis par une *URIref*, ils sont représentés par une constante, c'est-à-dire par une chaîne de caractères. La valeur est dès lors appelée un *littéral*.

Pour illustrer ces propos, le fait que Alice a un enfant dont le prénom est Bob est pris en exemple. La représentation de ce fait en *RDF* peut être décrite par le graphe de la figure 2.2.

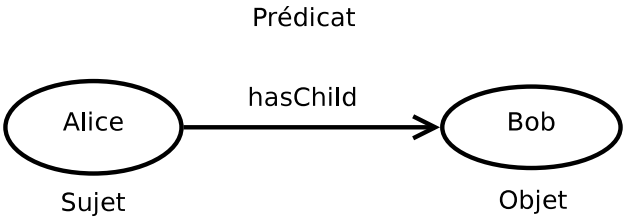

FIG. 2.2 – Alice a un enfant dont le prénom est Bob

L'expression sous la forme d'un triple constitue une autre représentation d'une assertion. Chaque triple reprend dans l'ordre le *Sujet*, *Prédicat* et l'*Objet*. Le triple [*Alice, hasChild, Bob*] correspond aux deux noeuds et à l'arc de la représentation précédente. Le graphe et les triples représentent la même information.

Dans l'exemple explicité précédemment, la propriété *hasChild* pourrait être identifiée par l'*URIref* `http://exemple.com/#has_child`. Cette *URIref* définit l'espace de nommage des exemples de ce travail au préfixe *hasCchild*

Des organismes internationaux ont défini des concepts clefs dans des domaines particuliers afin de proposer des descriptions standardisées des ressources. L'accès à ces concepts se fait par les espaces de nommage fixés par les organismes. Deux de ces organismes seront présentés dans la suite de ce mémoire.

RDF Schéma

Le lecteur retrouvera les idées ayant permis la rédaction de cette partie concernant *RDF Schéma* sur le site internet du *W3C* [BV04].

L'expressivité du langage *RDF* est limitée car les propriétés ne présentent que des relations entre des ressources. Il est impossible de décrire ces propriétés ni même de spécifier des relations entre des propriétés et d'autres ressources. Le *langage de description du vocabulaire RDF, RDF Schéma* est une extension du langage *RDF*. Celle-ci offre la possibilité de définir des classes et des propriétés pouvant être utilisées pour décrire des classes, des propriétés et d'autres ressources.

Il existe une différence majeure entre les classes de *RDF Schéma* et l'approche traditionnelle Orientée Objet telles qu'elles sont exprimées en Java par exemple. En Java, une classe est une collection de propriétés que possèdent les instances de cette classe tandis que *RDF Schéma* décrit un ensemble de propriétés s'appliquant aux classes.

Le langage de description du vocabulaire RDF fournit la possibilité de décrire des entités via des sous-classes et des superclasses. Toute propriété définie pour une superclasse est aussi valable pour toutes ses sous-classes. Comme il a été expliqué antérieurement, les propriétés spécifient une relation entre un *Sujet* et un *Objet*, mais il existe aussi des sous propriétés et des super propriétés telles que le domaine et l'ensemble de résultats. Tout le vocabulaire est détaillé dans les spécifications de *RDF Schéma* sur le site du *W3C* [BV04] et il est regroupé dans l'espace de noms appelé *rdfs* dont l'*URIref* est `http://www.w3.org/2000/01/rdf-schema#`.

La figure 2.3 représente graphiquement la modélisation de la famille avec *RDF Schéma*. Les noeuds de ce graphe sont les ressources *RDF* et les arcs les relations entre ceux-ci. Les étiquettes des arcs référencent quant à elles les propriétés entre les ressources.

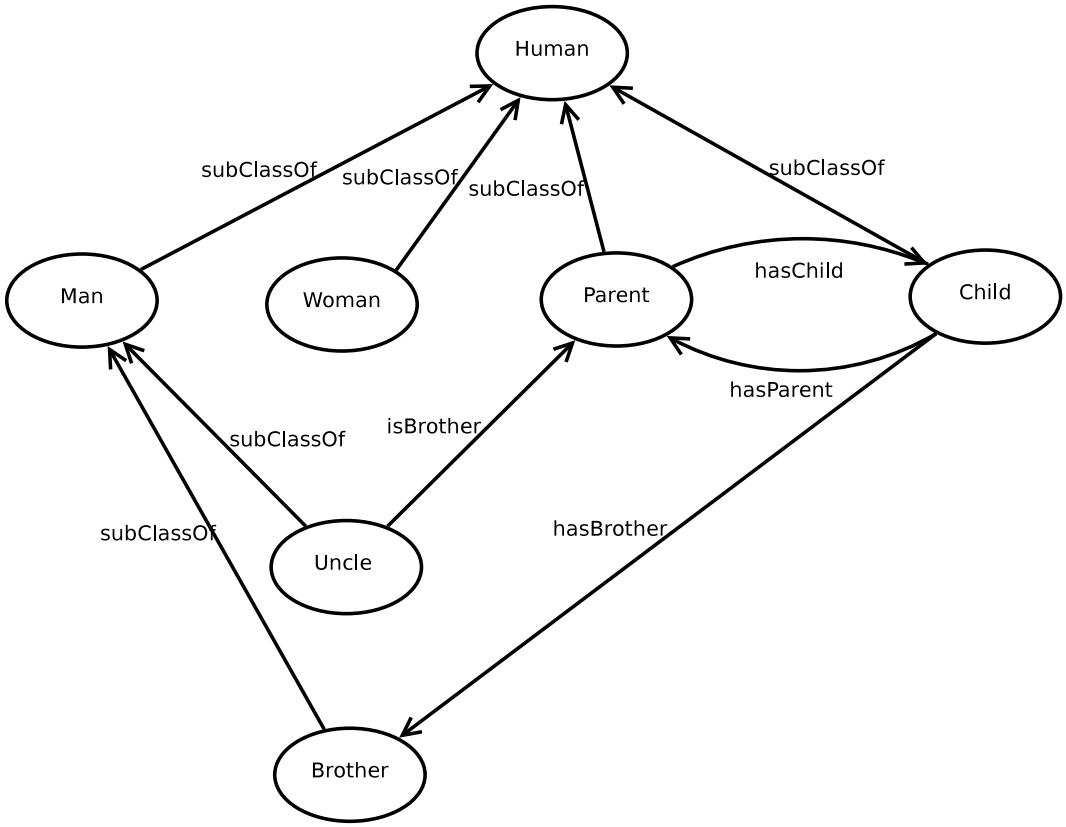

FIG. 2.3 – La représentation graphique de la modélisation de la famille avec *RDF Schéma*

RDF Schéma ne permet pas la généralisation des données sur le *Web* car son vocabulaire n'est pas suffisant. D'autres formes plus riches de description de données seront développées dans la suite de ce mémoire.

2.1.3 Vers l'harmonisation des ressources

Depuis l'apparition des technologies du *Web sémantique*, certains organismes ont pris l'initiative de proposer des standards de description d'éléments décrivant des domaines particuliers. L'objectif de cette initiative est de fournir le moyen d'harmoniser la description des ressources présentes sur *Internet*. Le lecteur trouvera ci-dessous, plus de détails concernant deux de ces organismes : *Dublin Core Metadata Initiative (DCMI)* et *Publishing Requirements for Industry Standard Metadata (PRISM)*.

Dublin Core Metadata Initiative

Dublin Core Metadata Initiative (DCMI) est une organisation qui promeut l'adoption de standards de méta données et qui définit des méta données pour décrire des ressources afin d'avoir plus de systèmes intelligents de découverte de l'information [CMI08]. Cet organisme explique sur son site internet [CMI08], que la mise sur pieds de tels standards permet de trouver, d'échanger et de gérer l'information. Ce modèle de données comprend

des éléments de descriptions formelles, intellectuels et relatifs à la propriété intellectuelle. L'ensemble des standards est simple et documenté afin de favoriser son utilisation par un grand nombre de personnes.

Publishing Requirements for Industry Standard Metadata

PRISM est un ensemble de spécifications de méta données facilitant la gestion et l'agrégation du contenu de différents magazines, de journaux, de catalogues, de livres, etc. Différentes compagnies proposant des éditoriaux sur le *Web* se sont rendues compte de la nécessité de formaliser les concepts dans le domaine de l'édition et ont proposé ces standards. Le lecteur trouvera plus d'informations concernant *PRISM* sur le site internet du groupe de travail [fISM08].

Un grand pas en avant

RDF, *RDF Schéma* et le travail de standardisation des concepts des différents organismes constituent un grand pas en avant dans la démarche visant à normaliser la description des ressources. Avec ces technologies, un bon nombre de besoins ont été identifiés et satisfaits. Cependant, le manque d'expressivité du langage *RDF* et *RDF Schéma* constitue encore un obstacle important à la généralisation des descriptions et rend difficile l'automatisation de l'échange des données entre les agents *Web*. L'introduction des ontologies et plus particulièrement de *Ontology Web Language* vient pallier ce problème en permettant d'affiner la description des entités.

2.2 Ontology Web Language (*OWL*)

Cette partie de ce mémoire porte sur le langage *OWL* qui implémente la couche *Ontology vocabulary* (voir schéma 2.1 page 13). Le terme "ontologie" est utilisé en philosophie pour désigner l'étude de tout ce qui existe. En informatique, il spécifie une conceptualisation en fournissant une description des concepts et leurs relations [Gru93].

Les ontologies sont représentées par des langages d'ontologie. Le *Web sémantique* propose *OWL*, un langage utilisant et enrichissant les notions de *RDF* et de *RDF Schéma*. De nouveaux concepts leur sont ajoutés afin d'augmenter leurs expressivités. Dès lors, les entités peuvent être décrites comme dans le monde réel et leur échange automatique sur le *Web* devient possible.

Dans la littérature, les concepts de *RDF* et de *RDF Schéma* utilisés pour modéliser les ontologies sont considérés comme faisant partie du langage *OWL*. Dans la suite de ce travail, le recours à cette simplification facilitera la compréhension du lecteur.

Dans cette section, les limitations de *RDF* et de *RDF Schéma* sont abordées. Ensuite, les ontologies sont définies et décrites avant de s'attarder à l'*Ontology Web Language*. Les idées exposées sont principalement tirées du document du *W3C* [vHM04] excepté lorsqu'une autre source est citée.

2.2.1 Les limitations de *RDF* et de *RDF Schéma*

Les limitations de *RDF* et de *RDF Schéma* viennent du fait que ces technologies ne permettent pas d'exprimer certains concepts. L'utilisateur qui décrit des ressources du monde réel a très souvent besoin de relations entre les classes et entre les propriétés mais celles-ci ne sont pas toujours définies dans ces deux langages. Les insuffisances de *RDF* et de *RDF Schéma* ont été identifiées par Jorge Cardoso [Car06] et sont reprises ci-dessous.

1. Les prédicats n'expriment que des relations binaires
2. Certaines relations entre les classes et les propriétés ne peuvent être énoncées. Il est intéressant de définir des propriétés telles que l'inverse, la transitivité, le symétrique, la disjonction, l'union et l'intersection. D'ailleurs, l'équivalence entre des descriptions construites par différents groupes de travail doit pouvoir être spécifiée.
3. Avec *RDF* et *RDF Schéma*, il est possible d'exprimer différentes valeurs d'une propriété mais pas de spécifier l'inexistence d'autres valeurs pour cette classe. Par exemple, l'énumération d'un classe *Genre* spécifiant le sexe d'un être humain n'est constituée que de 2 valeurs : *Homme* et *Femme*.
4. La cardinalité des propriétés ne peut pas être fixée. Pour certaines propriétés ou pour certaines classes, le nombre de valeurs d'instances doit être limité. Par exemple, la cardinalité de la propriété liant une classe *Enfant* à une classe *Père* est de 1.

Les insuffisances des représentations fournies par *RDF* et *RDF Schéma* peuvent être comblées par les ontologies. Celles-ci mettent à disposition une vaste gamme de concepts pour la description des classes et des propriétés d'un domaine.

2.2.2 Les ontologies

Selon les idées [LÖ08] reprises par Tom Gruber [Gru08], *une ontologie définit un ensemble de primitives représentatives qui modélisent un domaine de connaissances.* Les primitives dont il parle sont des classes (ou des ensembles), des attributs (ou des propriétés) et des relations (ou des relations entre les membres des classes). Entre certaines de ces primitives représentatives, des concepts de restrictions de valeurs, d'états disjoints, de cardinalités de classes et de propriétés, etc. peuvent être définis.

Les ontologies sont implémentées dans des langages permettant l'abstraction des structures de données. De ce fait, elles sont utilisées pour modéliser de nombreux domaines, sans tenir compte de la manière dont les données sont stockées. Il existe des langages standards (*OWL, Cycl, Gellish etc.*) ainsi que des outils commerciaux et open sources (*Protégé, Gene, KaOn, etc.*) pour créer et gérer les ontologies.

A partir d'une ontologie, il est possible d'appliquer des techniques mathématiques afin de trouver de nouvelles relations entre les éléments. Ces techniques combinent les relations de classes, de sous-classes et de superclasses avec les restrictions et les contraintes sur celles-ci pour produire de nouvelles connaissances. Les règles d'inférences utilisées sont génériques et se basent sur la logique des descriptions abordée ultérieurement dans ce travail. La généricité de raisonnement permet le recours à n'importe quel outil automatique pour traiter une ontologie. Parmi ces outils, se retrouvent *FaCt (Fast Classification of Terminologies)*, *RACER (Renamed A-box and Concept Expression Reasoner)* et *Pellet.* Il est important de noter que l'utilisation d'un outil se fait à condition que celui-ci soit compatible avec le langage d'implémentation de l'ontologie.

2.2.3 Le langage *OWL*

Le langage *OWL* est l'implémentation des principes spécifiés par les ontologies. Il se base sur *RDF* et *RDF Schéma* auxquels il ajoute du vocabulaire dont la signification est formalisée. Par la richesse de son expressivité, *OWL* n'étend pas seulement *RDF* et *RDF Schéma*, mais il ajoute un support de raisonnement capable d'inférer de nouvelles connaissances. L'inférence respecte les principes énoncés par la logique des descriptions. Tout comme les technologies qu'il complète, le langage *OWL* se transmet sur le *Web* via une extension du langage *XML.*

OWL est composé de 3 sous langages se différenciant par leur degré d'expressivité ;

1. *OWL Lite :* Ce sous langage supporte des classifications se basant sur les dépendances hiérarchiques entre les concepts. De plus, des contraintes simples sont exprimables. Il est non ambigü et le raisonnement sur les relations est décidable ; ce qui signifie que tout raisonnement sur une description se termine dans un intervalle de temps fini. Cependant,*OWL Lite* est peu expressif.
2. *OWL DL :* L'expressivité des descriptions est augmentée sans devoir sacrifier la décidabilité du raisonnement. Dès lors, toutes les inférences possibles sont prises en compte. Ce langage se base sur la logique des descriptions qui est la partie décidable

de la logique du premier ordre. L'utilisation d'outils adaptés au langage permet le raisonnement automatique sur les éléments décrits.

3. *OWL Full :* Avec ce sous langage, il est possible de tout exprimer au détriment de la garantie de calcul. Le raisonnement automatique ne prend pas en compte toutes les caractéristiques de ce sous langage afin d'éviter les situations d'indécidabilité.

Grâce à *OWL*, l'expressivité du schéma de la famille modélisé avec *RDF Schéma* (voir la figure 2.3 page 18) peut être augmentée afin d'améliorer la correspondance avec le monde réel. Les concepts ajoutés font partie de la technologie de *OWL Lite* et sont exprimés dans le tableau 2.1 sous la forme de formules logiques.

Axiome *OWL*
Father $\equiv$ Parent $\sqcap$ Man
Mother $\equiv$ Parent $\sqcap$ Woman
minCardinality(hasChild) = 1
minCardinality(hasParent) = 2
minCardinality(isBrother) = 1
$\equiv$ *est l'équivalence entre les classes.*
$\sqcap$ *est l'intersection entre les classes.*
minCardinality est le nombre minimum d'instances liant les deux classes de la relation

TAB. 2.1 – Les axiomes *OWL* modélisant la famille et leurs significations

Toutes les constructions fournies par le langage *OWL* sont décrites en détails sur le site du *W3C* [vHM04].

2.2.4 Un langage adapté à la réalité

La conceptualisation à l'aide des ontologies et plus particulièrement au moyen du langage *OWL* rend la description des ressources *Web* de plus en plus proche de la réalité. Grâce aux nombreux concepts de ce langage, il est possible de fournir une description des entités très complètes. En outre, le raisonnement automatique sur les classes et les propriétés par le biais de techniques mathématiques permet l'ajout de nouvelles connaissances. Le raisonnement peut être poussé plus loin et ne pas seulement être exécuté sur des classes et des propriétés. Dès lors, il opère sur des valeurs des variables de relations définies entre les classes. L'inférence est effectuée sur des règles *SWRL* qui enrichissent la sémantique de *OWL* en travaillant sur des instances des classes.

2.3 Semantic Web Rule Language *(SWRL)*

Le langage *SWRL* implémente la couche *Logic* du modèle proposé par le *Web sémantique* (voir schéma 2.1 page 13). Cette technologie est basée sur la combinaison des sous langages *OWL DL* et *OWL Lite* avec *Unary/Binary Datalog RuleML*, un sous langage de *Rule Markup Language* [HPSB+04]. *Rule Markup Language* est un langage de règles exprimé à l'aide de *XML* et conçu pour les applications de E-commerce [Rul08]. *SWRL* n'est pas un concurrent de *RuleML*, c'est un de ses sous-ensembles reprenant les relations unaires et binaires construites à partir des classes *OWL*. Il faut noter que le raisonnement sur les ontologies *OWL* augmentées par les règles devient indécidable.

Les règles *SWRL* ne créent pas de nouveaux concepts mais elles construisent des relations entre les classes *OWL*. L'interprétation d'une règle infére de nouveaux faits à ajouter à la base des connaissances associée au domaine modélisé.

Cette section reprend les grands principes de *SWRL* et décrit la structure des règles du langage. Ensuite, la sémantique du modèle théorique et l'interprétation qu'il est possible de faire à partir de ces règles sont abordées. Pour terminer, différentes applications et l'apport de *SWRL* sont spécifiés.

La plupart des idées reprises dans cette partie sont tirées du document proposé par le *W3C* [HPSB+04] excepté lorsqu'une source différente est annoncée.

2.3.1 La description du langage

Le langage *SWRL* augmente l'expressivité de *OWL* en construisant des règles avec les concepts définis dans l'ontologie. Chaque règle ressemble à une clause de *Horn* et prend la forme d'une implication entre un antécédent (le corps) et un conséquent (la tête).

$$\text{Règle :} \qquad \text{conséquent} \leftarrow \text{antécédent}$$

Le conséquent et l'antécédent sont constitués d'une conjonction d'atomes composée de zéro, d'un ou de plusieurs atomes. Chacun des atomes de la règle est un concept défini en *OWL* ou en *SWRL*.

Dans la règle 2.3.1 les deux atomes *hasChild(?Parent, ?Child)* et *hasBrother(?Parent, ?Uncle)* forment l'antécédent. *hasUncle(?Child, ?Uncle)* constitue le conséquent. *Child, Parent* et *Uncle* sont les variables de la règle.

Règle 2.3.1 ***Def-hasUncle***

$$hasUncle(?Child, ?Uncle) \leftarrow hasChild(?Parent, ?Child) \wedge hasBrother(?Parent, ?Uncle)$$

Les différents types d'atomes pouvant faire partie d'une règle sont repris dans le tableau 2.2 page 24.

Type d'atome	**Signification**
C(x)	*C* fait référence à une description *OWL*.
P(x,y)	*P* est une propriété *OWL*. La valeur de *x* par la propriété *P* est *y*.
sameAs(x, y)	*sameAs* est une propriété définie en *SWRL* imposant à *x* d'être identique à *y*.
differentFrom(x,y)	*differentFrom* est une propriété définie en *SWRL* imposant à *x* d'être différent de *y*.
built-in(r, x, ...)	*built-in* est une propriété prédéfinie en *SWRL* dans laquelle la relation *r* est appliquée à ses arguments.
x et y sont des variables, des individus OWL ou des valeurs de données OWL. r est la relation prédéfinie entre la variable x et celles qui suivent dans la relation.	

TAB. 2.2 – Les types d'atomes du langage *SWRL*

Dans la règle 2.3.2, $Adult(?Person)$ et $Person(?Person)$ sont des descriptions *OWL*, $hasAge(?Person, ?Age)$ est une propriété *OWL* indiquant qu'une personne a un âge et $swrlb : greaterThanOrEqual(?Age, 18)$ est un *built_in SWRL* imposant à un âge d'être plus grand ou égal à 18.

Règle 2.3.2 ***Def-Adult***

$$\begin{aligned} Adult(?Person) \leftarrow\ & swrlb : greaterThanOrEqual(?Age, 18) \\ & \wedge\ hasAge(?Person, ?Age) \\ & \wedge\ Person(?Person) \end{aligned}$$

Les variables présentes dans ces atomes sont quantifiées universellement et leur portée est limitée à la règle. En outre, seules les variables apparaissant dans l'antécédent peuvent se retrouver dans le conséquent.

Les règles dont le conséquent est composé d'une conjonction d'atomes sont transformables en règles distinctes dans lesquelles le conséquent est atomique. Cette transformation doit suivre les principes énoncés par J. W. Lloyd [Llo87].

Chaque règle *SWRL* peut être identifiée par une *URIref*. Pour que cette règle soit considérée comme vraie, si les conditions spécifiées dans l'antécédent sont respectées, alors les conditions du conséquent doivent l'être aussi. Lorsque l'antécédent d'une règle est vide, il satisfait toutes les interprétations possibles (il est considéré comme vrai). Dès lors, le conséquent est toujours satisfait. Au contraire, un conséquent vide est traité comme faux et il ne peut être satisfait par aucune interprétation. Les conditions de satisfaction des éléments d'une règle sont abordées dans la partie concernant la sémantique du modèle théorique.

La syntaxe abstraite présentant les règles sous la forme d'une implication et dans laquelle les variables sont préfixées par un point d'interrogation (voir les règles 2.3.1 et 2.3.2)

est destinée aux utilisateurs de *SWRL*. Cependant la syntaxe concrète est basée sur la syntaxe *XML* de *OWL* et de *RuleML*.

2.3.2 La sémantique du modèle théorique

Analyser la sémantique du modèle théorique d'un langage, c'est comprendre la signification des éléments composant ce langage. La sémantique de *SWRL* est une extension de la sémantique du modèle de *OWL* qui fait correspondre des variables à des éléments du domaine.

Les règles décrites par le langage *SWRL* sont interprétables. Une règle est satisfaite par une interprétation si et seulement si chaque association satisfaisant l'antécédent satisfait aussi le conséquent. Une association est une correspondance entre les variables des atomes de la règle et une de leurs instanciations possibles. Pour un atome définissant un concept *OWL* les variables représentent des individus des classes, des types de données ou des valeurs littérales. Les atomes définis dans le langage *SWRL* sont des contraintes portant sur l'instanciation des variables.

Pour chacun des atomes composant l'antécédent et le conséquent d'une règle *SWRL*, le tableau des conditions d'interprétation (voir table 2.3 page 25) donne les conditions selon lesquelles l'interprétation de l'association peut être satisfaite.

Atome	**Condition d'interprétation**
C(x)	x est une instance de la classe C.
D(z)	d est une valeur de la donnée D.
P(x,y)	x est lié à y par la propriété P.
Q(x,z)	x est lié à z par la propriété Q.
sameAs(x, y)	x et y représentent le même individu.
differentFrom(x,y)	x et y représentent des individus différents.
built-in(r, z1,..., zn)	la relation r est évaluée à *true* avec les arguments *z1,..,zn*.
C est une description OWL DL, D est l'ensemble image défini en OWL DL, P est une propriété d'individu, Q est une propriété de type de données , f est une relation prédéfinie, x ,y sont des variables ou des identifiants référençant des individus OWL, et z est une variable ou une variable OWL évaluée (un littéral).	

TAB. 2.3 – Les types d'atomes d'une règle et les conditions d'interprétation

L'interprétation de la règle 2.3.1 (voir page 23) se fait comme suit. Si l'enfant d'Alice est Bob et si celle-ci à un frère prénommé Chris alors Chris est l'oncle de Bob.

2.3.3 L'apport de *SWRL*

L'intégration de la logique et d'un système de règles augmente considérablement les possibilités de raisonner sur les ressources décrites sur le *W.W.W*.

A l'heure actuelle, *SWRL* est utilisé sur le *Web* pour implémenter des applications incluant des règles concernant les affaires. Ce langage permet de prendre en compte les préférences des utilisateurs et de les combiner avec les besoins des clients. Sur le *Web*, de nombreuses applications développées décrivent des règles utilisées par des agents intelligents afin de compléter la base des connaissances. A partir de ces règles, ces agents sont capables de raisonner dans le but de communiquer, de négocier, d'exécuter des contrats. Le langage *SWRL* sert aussi à fixer des prix en fonction de différents facteurs tels que le profil du client (régularité d'achat, solvabilité, etc.), les conditions de vente, les délais de réapprovisionnement du produit, etc. Il existe beaucoup d'autres applications de *SWRL* dans le monde de l'E-Business qui ne seront pas détaillées dans ce travail.

Toutefois, les utilisateurs de *SWRL* sont confrontés à certains problèmes fixant les limites de ce langage. Certaines propriétés sont inexprimables et le raisonnement sur le langage *SWRL*, tout comme sur la logique du premier ordre est indécidable.

Dans le chapître suivant, la logique de description est abordée afin de mieux comprendre les limites de *SWRL* ainsi que le processus d'inférence s'exécutant à partir des règles décrites dans ce langage et permettant la découverte de nouveaux faits.

2.4 Le Web sémantique : le bilan

Les idées reprises dans cette partie sont inspirées du livre [AS06] de H. P. Alesso et C. F. Smith.

Les applications supportant les technologies du *Web sémantique* sont non seulement capables de comprendre l'information mais surtout de raisonner sur les données afin de découvrir automatiquement de nouvelles connaissances. Les technologies du *Web sémantique* ne visent pas seulement les pages *Web*. Elles améliorent aussi les échanges entre des bases de données modélisées selon des schémas différents ou encore entre des programmes écrits dans des langages distincts. L'apport principal de ces technologies vient du fait qu'elles peuvent exprimer n'importe quelle relation entre les concepts d'un domaine modélisé. De nombreux organismes ont mis sur pieds des outils capables de manipuler ces différentes technologies car le nombre d'applications développées à partir de celles-ci ne cesse d'augmenter.

Avec ces technologies, la nouvelle vision du *Web* proposée par le *Web sémantique* visant à rendre l'échange des données automatique devient possible. Ces technologies sont utilisées dans des domaines particuliers dont certains sont détaillés ci-dessous :

Les services du *Web sémantique* : Les *Services Web* sont des programmes informatiques capables de communiquer et d'échanger des données entre des applications et des systèmes hétérogènes dans des environnements distribués. L'interaction avec les applications développées par d'autres est souvent difficile. Les technologies du *Web sémantique* rendent ces services interopérables en fournissant un langage commun, compatible et échangeable de description des services. En outre, le contenu du *Web* pourrait être combiné avec ces services afin que des agents *Web* intelligents raisonnant sur des règles business automatisent la découverte, l'invocation et la composition d'information. *Semantic Markup for Web Services (OWL-S)*, construit à partir de *OWL*, fournit les concepts pour décrire les services.

La recherche sémantique : Le nombre de pages référencées sur le *Web* ne fait qu'augmenter rendant le travail des moteurs de recherches de plus en plus difficile. En outre, les moteurs de recherches actuels ne peuvent répondre à certaines requêtes. Par exemple, il leur est impossible de trouver le moyen de transport le moins cher pour aller de Namur à Bruxelles. Afin d'être performants, les moteurs de recherche doivent interpréter l'information, filtrer le résultat de la recherche et ne garder que les pages répondant réellement à la requête de l'utilisateur. Les moteurs de recherche réalisés sur base des idées proposées par le *Web sémantique* peuvent interpréter les concepts et faire des recoupements logiques sur bases des relations entre ces concepts. Leur fonctionnement n'est plus seulement basé sur la recherche d'un mot-clef mais la sémantique des mots est prise en compte et les résultats non pertinents sont mis de côté.

L'apprentissage en ligne : L'apprentissage en ligne à grande échelle nécessite le déploiement d'un environnement collaboratif par les organismes éducatifs. Ceux-ci doivent mettre à disposition des élèves une application supportant l'interopérabilité des différents systèmes. Le *Web sémantique* fournit le moyen de conceptualiser la sémantique

des éléments du domaine via les ontologies, d'exprimer une syntaxe de communication commune standardisée et d'intégrer le contenu éducatif à grande échelle. Avec ces technologies, l'apprentissage à distance devient possible car les organismes éducatifs peuvent travailler ensemble même s'ils sont localisés dans des endroits éloignés.

La sémantique de la bio-informatique : La bio-informatique désigne le travail collaboratif de recherche impliquant des informaticiens, des mathématiciens, des physiciens et des biologistes. Ceux-ci s'interrogent sur les grands problèmes de la biologie et tentent d'apporter des solutions en combinant leurs savoirs. Aux Etats-Unis, il existe une initiative visant à appliquer les standards du *Web sémantique* aux soins de santé dans l'intention de faciliter l'accès à ces données aux différentes institutions. La mission du groupe *HCLSIG (The Semantic Web for Health Care and Life Sciences Interest Group)* est d'améliorer la collaboration, la recherche et le développement entre le monde scientifique et les organismes s'occupant des soins de santé. Le succès de ce défi nécessite la définition d'un vocabulaire commun et une modélisation via les ontologies afin de déployer un système interopérable intégrant les données issues des différents domaines. Le lecteur intéressé peut trouver de l'information complémentaire sur le site du groupe `http://www.w3.org/2001/sw/hcls/`.

L'intégration d'applications d'entreprise : Les technologies du *Web sémantique* fournissent des outils pour intégrer les applications dans les entreprises. Les ontologies aident les développeurs et les administrateurs dans les tâches d'analyse, de développement, de déploiement et d'exécution de l'application. Les ontologies permettent d'harmoniser les modèles conceptuels et la description des services. Elles permettent aussi de décrire les composants d'un domaine hétérogène et de formaliser les concepts ainsi que les relations entre ceux-ci. A l'exécution, les ontologies sont couplées avec un moteur d'inférence s'intégrant à toutes les applications et permettant d'en déduire de nouvelles connaissances.

La base de connaissances : Les technologies du *Web sémantique* peuvent être utilisées afin de créer des systèmes de connaissances capables d'acquérir, d'organiser, de traiter, de partager et d'utiliser les connaissances. L'entreprise *Cycorp* `http://www.cyc.com`) rassemble dans des ontologies et des bases de données les connaissances du sens commun (normes, valeurs symboliques, etc.). Elle fournit également un programme capable de raisonner sur ces données afin de venir en aide aux applications d'intelligence artificielle.

Après ce bilan, le lecteur se rend compte que les technologies proposées par le *Web sémantique* sont belle et bien d'actualité. Ce phénomène est marqué aux Etats-Unis par le lancement de jeunes entreprises (Hakia, Powerset, etc.) dont l'activité principale est de mettre au point un moteur de recherche se basant sur la sémantique des mots-clefs.

Chapitre 3

Les technologies d'évaluation de règles

Alors que la partie précédente présentait certains langages proposés par le *Web sémantique*, ce chapître se penche sur l'activité de raisonnement qu'il est possible de faire à partir de ces langages ainsi que sur le processus d'évaluation des règles. Dans la première partie, la logique de description est présentée. Elle fournit les explications nécessaires à la compréhension de la logique se cachant derrière les technologies *RDF*, *OWL* et *SWRL*. La logique de description détaille le processus d'inférence permettant de découvrir de nouvelles connaissances à partir des faits modélisés par ces langages. Ensuite, la partie sur le chaînage présente le chaînage avant et le chaînage arrière qui sont deux méthodes permettant d'évaluer un ensemble de règles et de vérifier la réalisation d'un but particulier. Et pour terminer, la transformation en *ensembles magiques* et l'algorithme *Rete* sont décrits. Ces deux méthodes sont des optimisations de l'algorithme "naïf" de chaînage avant.

3.1 La logique de description

La plupart des idées reprises dans cette partie sont tirées de [Wal07] sauf lorsqu'une autre source est citée.

La logique de description décrit une famille de langages qui sont des sous-ensembles de la logique du premier ordre adaptés à la représentation des réseaux sémantiques. *RDF* est un réseau sémantique car cette technologie permet la modélisation du domaine de connaissances sous la forme d'un graphe.

La logique de description est utile pour représenter les connaissances formellement et pour raisonner à partir de celles-ci afin de découvrir de nouveaux faits. Les connaissances sont exprimées par l'intermédiaire d'une variante du calcul des prédicats et il est possible de raisonner sur celles-ci en appliquant les règles de la logique. Le langage *OWL* est basé sur la logique de description. Elle fournit les formalismes nécessaires à l'expression des concepts de *OWL* et surtout de *OWL-DL*. En outre, la logique de description sert de base au raisonnement effectué sur les concepts définis avec le langage *OWL*.

La logique de description se caractérise par trois niveaux de description des concepts :

1. Les concepts sont structurables en hiérarchie de superclasses et de sous-classes appelées relations de subsomptions. Les sous-classes héritent des relations de la superclasse.
2. Les individus sont classifiables et il est possible de déterminer quels concepts sont les instances d'une classe spécifique.
3. Entre les concepts, des relations spécifiques sont définies.

Ces trois niveaux de description des concepts rendent possible la modélisation de domaine de connaissances très variés.

3.1.1 La description du langage

Dans cette section la représentation syntaxique de la logique de description est abordée.

Les structures définies dans la logique de description sont très similaires à celles des formules de la logique du premier ordre. La description des concepts du domaine se font via des assertions logiques. Celles-ci ne contiennent pas de variables mais seulement des symboles de base définis dans deux alphabets disjoints repris ci-dessous.

1. Les concepts atomiques sont les symboles représentant des prédicats unaires qui expriment les concepts.
2. Les rôles atomiques sont les symboles représentant des prédicats binaires qui expriment les relations entre les concepts.

Tous les langages de description reprennent ces concepts et ces rôles atomiques afin de former la description des concepts. La combinaison des deux alphabets permet la construction de deux types de description : la description élémentaire et la description complexe. La première reprend les concepts et les rôles atomiques tandis que la seconde est obtenue en combinant les concepts et les rôles atomiques au moyen de constructeurs fournis par le langage. Les constructeurs varient en fonction de la définition du langage de description. Au plus il y a de constructeurs, au plus le langage est expressif mais au plus le raisonnement est difficile et risque d'être indécidable. Les constructeurs disponibles dans un langage en définissent son nom. Le langage $\mathcal{ALCN}$ reprend les concepts clefs décrits dans beaucoup de langages de description. $\mathcal{AL}$ fait référence au fait que ce langage comprend des concepts et des relations atomiques. $\mathcal{C}$ désigne le complément pour exprimer la négation et $\mathcal{N}$ les restrictions de cardinalité. La description et la sémantique des différents constructeurs compris dans le langage de description $\mathcal{ALCN}$ sont repris dans le tableau 3.1 (voir page 31).

C[1]	Description	Sémantique
$\top$	Le concept universel. C'est l'ensemble de tous les concepts du domaine.	Δ
$\bot$	Le concept "ground". Cet ensemble ne contient pas de concept.	$\emptyset$
$\neg C$	La négation.	$\Delta - C$
$C \sqcap D$	L'intersection. Ce concept est équivalent à C(x) $\wedge$ D(x) exprimé par la logique du premier ordre.	$C \cap D$
$C \sqcup D$	L'union. Ce concept est équivalent à C(x) $\vee$ D(x) exprimé avec la logique du premier ordre.	$C \cup D$
$\forall R.C$	La restriction de valeur. Tous les individus qui ont une relation de R vers R sont inclus dans C.	$a \in \Delta \mid \forall b.(a,b) \ \in R \rightarrow b \in C$
$\exists R.C$	La quantification existentielle identifie certains individus de la relation R qui contiennent le concept C.	$a \in \Delta \mid \exists b.(a,b) \ \in R \wedge b \in C$
$\leq n\, R$	Le nombre d'individus maximum qui participent à la relation R.	$a \in \Delta \mid \|\|\{(a,b) \ \in R\}\| \| \leq n$
$\geq n\, R$	Le nombre d'individus minimum qui participent à la relation R.	$a \in \Delta \mid \|\|\{(a,b) \ \in R\}\| \| \geq n$

TAB. 3.1 – La description du langage $\mathcal{ALCN}$

OWL-DL est fondé sur $\mathcal{SHOIN}(\mathcal{D})$ *DL*. $\mathcal{S}$ est l'abréviation de $\mathcal{ALC}_{\mathcal{R}+}$. Dans ce langage, la transitivité entre les rôles est ajoutée à $\mathcal{ALC}$. $\mathcal{H}$ reprend les rôles hiérarchiques, $\mathcal{O}$ les nominaux, $\mathcal{I}$ les rôles inverses, $\mathcal{N}$ les restrictions de quantification et $\mathcal{D}$ fait référence au type de données. Les langages $\mathcal{ALCN}$ et $\mathcal{SHOIN}(\mathcal{D})$ sont très similaires, et de ce fait, $\mathcal{ALCN}$ va servir de base à l'explication des différents concepts décrits dans la suite de ce mémoire.

3.1.2 La représentation des connaissances

En plus de fournir la syntaxe décrivant les concepts d'un domaine, la logique de description construit des bases de connaissances avec lesquelles il est possible d'interagir.

Dans la logique de description, la représentation des connaissances se fait à deux niveaux. Le premier est le niveau terminologique, appelé le niveau *TBox*, décrivant les concepts du domaine ainsi que leurs rôles. De nouveaux concepts peuvent être établis par la combinaison de concepts existants. Les *TBox* conceptualisent un domaine et ne risquent pas de changer. Par contre, le niveau *ABox* est dynamique car il exprime des faits propres à une situation particulière. Les assertions *ABox* décrivent les concepts du niveau *TBox* en reliant les individus aux concepts et aux rôles définis à ce niveau. De manière plus

[1] Le constructeur

générale, le niveau *TBox* correspond aux ontologies et le niveau *ABox* aux assertions *RDF*.

Les tableaux 3.2 (voir page 32) illustrent la représentation des connaissances de la modélisation d'une famille.

TBox
Woman $\equiv$ Person $\sqcap$ Female
Man $\equiv$ Person $\sqcap\neg$ Woman
Mother $\equiv$ Woman $\sqcap \exists hasChild.Person$
Father $\equiv$ Man $\sqcap \exists hasChild.Person$
Grandmother $\equiv$ Mother $\sqcap \exists hasChild.Parent$
MotherWithManyChildren $\equiv$ Mother $\sqcup \geq 3$ hasChild

ABox
Person(Paul)
Man(Paul)
Person(Alice)
hasChild(Paul, Alice)
Father(Paul)
Woman(Alice)

TAB. 3.2 – Les connaissances relatives à la famille

Les interactions avec la base de connaissances sont de deux types : soit l'utilisateur exécute des opérations d'ajout, soit il demande de l'information. Les requêtes de l'utilisateur sont construites selon la représentation des informations dans la base de connaissances.

Le lecteur intéressé par de plus amples explications concernant les spécificités de la logique de description peut consulter [BCM+03]

3.1.3 L'inférence sur les connaissances

Il a été vu précédemment que les connaissances sont représentées aux niveaux *TBox* et *Abox*. Une base de connaissances est plus qu'un conteneur, c'est un système expert capable de répondre à des questions concernant les connaissances qu'il contient. L'activité de raisonnement est réalisée par les mécanismes de l'inférence logique. Une base de connaissances est similaire à un ensemble d'axiomes de la logique des prédicats. Ces axiomes contiennent des connaissances rendues explicites par l'inférence. Les connaissances représentées par les *ABox* et les *Tbox* sont différentes. Le raisonnement sur ces connaissances doit donc être effectué différemment.

Pour les *TBox*, il y a quatre types d'inférences à prendre en compte [Wal07].

1. La subsomption : $T \vdash C \sqsubseteq D$ exprime le fait que le concept D est plus général que C dans la *TBox* T. Cela signifie que l'ensemble C est un sous-ensemble de D. L'inférence permet la découverte de nouvelles relations entre les concepts qui n'avaient pas été définies à la base.
2. La satisfaisabilité : Lorsqu'un nouveau concept est défini dans la base des connaissances, est-il consistant ? Un concept est consistant s'il est possible de construire un individu correspondant à ce concept. Un concept consistant est un concept satisfaisable. Dans le cas contraire il est insatisfait et $T \vdash C \sqsubseteq \bot$.

3. L'équivalence : L'inférence teste si deux concepts sont équivalents. Cela permet de supprimer certaines redondances et des ambiguïtés dans la base de connaissances. $T \vdash C \equiv D$ exprime que C et D sont équivalents dans la $TBox$ T.
4. La disjonction : L'inférence vérifie que deux concepts sont disjoints c'est-à-dire qu'il n'y a pas de relation à inférer entre eux. $T \vdash C \sqcap D \sqsubseteq \bot$.

Les connaissances $ABox$ portent sur les individus. Il y a quatre types d'inférences réalisables sur ces connaissances [Wal07].

1. La vérification d'instance : L'inférence vérifie qu'un individu correspond à un concept spécifique défini dans une $TBox$. $A \vdash C(a)$ signifie que l'instance a appartient au concept C.
2. L'extraction de données : Il s'agit de trouver toutes les instances d'un concept dans la base de connaissances. $a \in A | A \vdash C(a)$.
3. La réalisation : Le concept le plus spécifique doit être trouvé pour chaque instance. $A \vdash C(a)$ et $T \vdash \bot \sqsubseteq C$.
4. La cohérence : Chaque concept de la base de connaissances $TBox$ admet au moins un individu dans les connaissances $ABox$. $\forall C \in T \exists a \in A | A \vdash C(a)$.

Le moteur d'inférence interprétant le domaine modélisé par les ontologies utilise les huit types d'inférences définis ci-dessus.

3.2 Le chaînage

Les idées et les terminologies à la base de la rédaction de la partie concernant le chaînage avant et arrière sont tirées de [RN03].

Le chaînage est l'activité de déduction logique appliquée à des connaissances exprimées sous la forme de clauses de *Horn*. Cette activité nécessite une base de connaissances des faits considérés comme vrais, une base de règles modélisant les savoirs du domaine et la spécification d'une solution à vérifier. L'objectif du chaînage est de traiter les données pour vérifier la réalisation d'un but représentant un fait particulier. Pour exploiter les règles, l'inférence peut être réalisée soit en chaînage avant, soit en chaînage arrière.

Avant de présenter les deux types de chaînage plus en détails, il est intéressant de préciser quelques concepts permettant de mieux comprendre le processus d'inférence.

L'*Instanciation Universelle* est une règle d'inférence utilisée lors de l'évaluation d'un atome quantifié universellement. *Il est possible d'inférer n'importe quel fait obtenu en substituant la variable d'un atome universellement quantifié par un terme ground (un terme sans variable)* [RN03].

La substitution $SUBST(\theta, \alpha)$ montre le résultat de l'application de la substitution θ à la clause α.

$$\frac{\forall\ v\ de\ \alpha}{SUBST(\{v/g\}, \alpha)}$$

v est une variable et g est le symbole *ground*.

L'atome $Man(X)$ est pris en exemple. Si $Man(john)$ est le fait associé à cet atome, la substitution vaut $SUBST(\{X/john\}, Man(X))$.

Les substitutions sont fixées en unifiant les atomes. A partir de deux atomes, l'unification retourne l'unificateur s'il existe.

$$UNIFIER(p, q) = \theta\ tel\ que\ SUBST(\theta, p) = SUBST(\theta, q)$$

Par exemple,

$$UNIFIER(hasBrother(Brian, x),\ hasBrother(Brian, Mary)) = \{x/Mary\}$$

$$UNIFIER(hasBrother(Brian, x),\ hasBrother(y, Mary)) = \{x/Mary, y/Brian\}$$

$$UNIFIER(hasBrother(Brian, x),\ hasBrother(x, Mary)) = fail$$

L'inférence sur les règles est guidée par la règle du *Generalized Modus Ponens*. Cette règle s'applique sur des atomes avec des variables *universellement quantifiées*. Elle est définie dans [RN03] comme suit.

Soit les clauses p_i, p'_i et q, pour lesquelles il existe une substitution θ telle que $SUBST(\theta, p'_i) = SUBST(\theta, p_i)\ \forall i, alors$

$$\frac{p'_1, p'_2, ..., p'_n, (p_1 \wedge_2 \wedge ... \wedge p_n \rightarrow q)}{SUBST(\theta, q)}$$

Dans $SUBST(\theta, q)$, θ est une substitution constituée d'une liste de substitution de chaque variable obtenue par les *Instanciation Universelle*.

La règle suivante est prise en exemple :

$$\forall X\ Father(X) \leftarrow Man(X) \wedge Parent(X)$$

Les différents faits sont $Man(john)$ et $Parent(john)$.
La subsitution θ vaut $SUBST(\{X/john\}, Father(X))$.

3.2.1 Le chaînage avant

Le chaînage avant est la procédure naturelle d'inférence logique. Elle se fait à partir des prémisses qui constituent les corps des règles. Les faits de la base de connaissances servent de base à l'évaluation des prémisses. Lorsque les clauses de la prémisse d'une règle sont respectées, les faits de la conclusion de l'implication (la tête de la règle) peuvent être ajoutés à la base de connaissances. A chaque fois que l'évaluation d'une règle est effectuée, le processus d'évaluation des règles est recommencé. En effet, les nouveaux faits ajoutés peuvent constituer une nouvelle entrée pour une règle déjà évaluée. L'itération est répétée jusqu'à ce qu'une solution au but soit trouvée (en supposant qu'une seule réponse est souhaitée) ou alors si aucun nouveau fait ne peut plus être découvert. Lorsqu'il n'est plus possible d'inférer de nouveaux faits en exécutant à nouveau le processus de chaînage, cela signifie que tous les faits sont déjà contenus explicitement dans la base de connaissances. Dans ce cas, il est dit que le point fixe du processus d'inférence est atteint.

A partir de la base de connaissances, l'algorithme 1 de chaînage avant (voir page 36) proposé analyse les règles pour trouver la réponse à la requête exprimée sous la forme d'une clause atomique. Cet algorithme est la simplification de l'algorithme proposé dans le livre [RN03].

L'algorithme 1 évalue chaque règle à partir des faits connus de la base des connaissances. Lorsque les prémisses de cette règles sont satisfaites, l'algorithme évalue la conclusion de la règle et de nouveaux faits peuvent être ajoutés aux faits connus. Ce processus est répété jusqu'à ce que le but associé à l'évaluation de ces règles soit atteint ou lorsque plus aucun nouveau fait ne peut être découvert. S'il existe une solution à la requête (l'algorithme ne renvoie pas *false*), elle est de la forme d'une substitution fournissant les substitutions des variables. Cet algorithme est présenté afin de comprendre le concept de chaînage avant mais il est bon de noter qu'il n'est pas efficace.

Algorithm 1 : function **Fol-FC(BC, α)**

Require: BC est la base de connaissances et α la requête sous la forme d'une clause atomique
Ensure: **returns** une substitution ou *false*
{*nouveau* représente les nouveaux faits inférés}

repeat
  nouveau $\leftarrow \{\}$
  for chaque règle r dans BC **do**
    $q \leftarrow p_1 \wedge ... \wedge p_n$ un renommage de r
    for chaque ensemble de faits $p'_1, ... \wedge p'_n$
    tel que θ = UNIFIER($p_1 \wedge ... \wedge p_n$, $p'_1 \wedge ... \wedge p'_n$) existe
    {pour des $p'_1, ... \wedge p'_n$ de BC} **do**
      $q' \leftarrow SUBST(\theta, q)$ où q est la conclusion de la règle courante
      if *q'* n'est pas déjà dans BC ou dans *nouveau* **then**
        ajouter *q'* à *nouveau*
        $\phi \leftarrow$ UNIFIER(q',α)
        if ϕ n'échoue pas **then**
          return ϕ
        end if
      end if
    end for
    ajouter *nouveau* à BC
  end for
until *nouveau* est vide
return *false*

Afin d'illustrer l'algorithme du chaînage avant, les règles suivantes sont utilisées.

Règle 3.2.1 *Def-hasUncle*

$$hasUncle(x, y) \leftarrow hasChild(z, x) \wedge hasBrother(z, y)$$

Règle 3.2.2 *Def-hasBrother*

$$hasBrother(x, y) \leftarrow hasSibling(x, y) \wedge Man(y)$$

A ces règles sont associés les faits suivants :

$$hasChild(Alice, Bob)$$

$$hasSibling(Alice, Chris)$$

$$Man(Chris)$$

Il ne reste qu'à définir la requête à laquelle le processus de chaînage avant doit répondre. Dans c'est exemple, il faut vérifier que *Chris* est bien l'oncle de *Bob* ($hasUncle(Chris, Bob)$). L'algorithme évalue les règles l'une après. Dans la règle 3.2.2, la substitution des clauses de la prémisse avec les faits $hasSibling(Alice, Chris)$ et $Man(Chris)$ permet d'inférer le fait que *Chris* est le frère d'*Alice*. $hasBrother(Alice, Chris)$ est donc ajouté à la base de connaissances. Dès lors, l'algorithme doit être réexécuté car le nouveau fait n'est pas la solution au but à atteindre. L'ajout de ce fait permet l'évaluation de la règle 3.2.1. Dans cette règle la substitution des clauses de la prémisse avec les faits de la base de connaissances permet d'inférer le fait que *Chris* est l'oncle de *Bob*. L'algorithme est arrêté car ce nouveau fait est la solution à la requête initiale.

L'algorithme 1 du chaînage avant (voir page 36) n'est pas optimal. Les différents défauts identifiés dans le livre [RN03] sont détaillés ci-dessous :

Des correspondances inutiles avec la base de faits sont trouvées : Cet algorithme n'est pas très performant car il trouve toutes les substitutions possibles pour chaque clause de la prémisse des règles avec la base de connaissances. De ce fait, un bon nombre de ces substitutions lorsqu'elles sont combinées entre elles sont écartées car elles ne respectent pas toutes les autres clauses de la prémisse. Une amélioration possible à apporter à cet algorithme est de déterminer un ordre d'évaluation des clauses de la prémisse afin de minimiser le nombre de substitutions inutiles. Ainsi, les coûts d'évaluation sont diminués. Par exemple, si les atomes *Uncle(x)* et *hasBrother(x, john)* constituent les clauses de la prémisse d'une règle, il est plus judicieux de d'abord extraire tous les frères de *john* et puis de regarder lequel de ces frères est un oncle.

Le chaînage avant est incrémental : Lors de la découverte de nouveaux faits, l'algorithme est réexécuté. Une nouvelle évaluation des règles est réalisée à partir de tous les faits même si peu de connaissances ont été ajoutées. Ainsi toutes les substitutions possibles sont trouvées alors qu'un bon nombre de celles-ci ont déjà été identifiées lors des itérations précédentes. Pour éviter ce travail inutile, chaque nouveau fait inféré

dans une itération doit être dérivé au moins d'un nouveau fait inféré à l'itération précédente. En effet, toutes les inférences qui ne nécessitaient pas de nouveaux faits ont déjà été effectuées lors de la dernière itération. Dès lors, pour une itération t, une règle est évaluée seulement si la prémisse inclut une clause p_i unifiée avec un fait p'_i inféré à l'itération *t-1*.

L'algorithme *Rete*, présenté dans une des sections suivantes, tient compte de cette optimisation.

L'inférence de conclusions non pertinentes : Le chaînage avant produit des conclusions inutiles car toutes les règles sont évaluées. Toutes les inférences sont donc effectuées et de nouvelles conclusions sont tirées même si elles n'apportent pas d'information permettant d'atteindre l'objectif de l'évaluation. La solution est d'utiliser le chaînage arrière ou de sélectionner un sous-ensemble pertinent des règles. L'idée est de réécrire les règles en utilisant les informations liées à l'objectif à atteindre. Ce processus est décrit dans la théorie sur les *ensembles magiques* se basant sur le chaînage avant.

3.2.2 Le chaînage arrière

L'algorithme d'inférence par chaînage avant se base sur les faits connus tandis que le mécanisme de chaînage arrière s'exécute à partir du but à établir. Celui-ci enchaîne les règles afin de trouver les faits supportant ce but. Cet algorithme est utilisé en programmation logique, notamment dans les systèmes *Prolog*, pour exécuter les raisonnements automatiques.

L'algorithme de chaînage arrière s'effectue à partir de la requête initiale qui est le but à prouver. Pour ce faire, un arbre de recherche est construit dans lequel ce but est placé à la racine. L'algorithme sélectionne ensuite une règle dans la base des règles dont la conclusion s'unifie avec le but recherché. Pour chaque clause de la prémisse de cette règle, une branche de type *et* est ajoutée à l'arbre de recherche la reliant à la racine. Ces clauses deviennent les nouveaux buts à prouver et le processus est appelé récursivement sur chacune d'elles. La décomposition en sous buts continue et l'arbre de recherche grandit.

S'il y a dans la base de règles plusieurs règles dont la conclusion s'unifie avec le but recherché, chacune d'elles doit être reliée au but concerné par une branche *ou* avant d'être décomposée en sous buts. Chaque branche *ou* de l'arbre de recherche donne lieu à la recherche d'un solution indépendante. Cela rend l'algorithme non-déterministe.

Lorsqu'un but peut être unifié avec un fait de la base de connaissances, le processus de décomposition en sous buts s'arrête et ce sous but est prouvé. La substitution qui a permis l'unification est appliquée aux autres sous buts de l'arbre faisant partie de la solution indépendante. Si un noeud du graphe est partagé en plusieurs branches *ou*, les substitutions trouvées pour une branche ne sont pas propagées à travers les autres branches *ou* reliées à ce noeud.

L'objectif de départ est prouvé lorsque tous les buts d'une solution indépendante de l'arbre sont satisfaits. L'algorithme de chaînage arrière parcourt l'arbre de recherche en

profondeur d'abord.

La solution à la requête initiale est la composition des substitutions de tous les sous buts de la solution indépendante. La compositition des substitutions est identique à l'effet d'appliquer toutes les substitutions tour à tour.

$$SUBST(COMPOSE(\theta_1, \theta_2), p) = SUBST(\theta_2, SUBST(\theta_1, p))$$

L'exemple ayant servi à illustrer l'algorithme du chaînage avant peut aussi être résolu par l'algorithme d'inférence en chaînage arrière. Les règles, la requête et les faits considérés restent les mêmes. L'arbre de recherche de la figure 3.1 (voir page 39) montre comment le chaînage arrière peut être réalisé. Dans cet arbre, il n'y a que des branches *et* car chaque but à satisfaire ne s'unifie qu'avec la conclusion d'une seule règle de la base de règles.

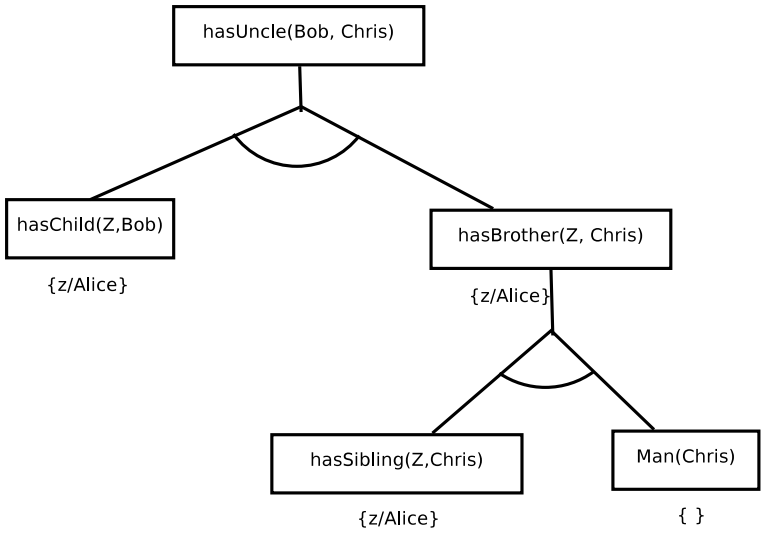

FIG. 3.1 – L'arbre de recherche vérifiant que *Chris* est bien l'oncle de *Bob*

Afin de justifier l'existence des deux techniques d'évaluation d'un ensemble de règles, il est intéressant de décrire les défauts majeurs de la méthode d'évaluation en chaînage arrière. Tout d'abord, il faut se rendre compte que certaines situations mettent en péril le bon déroulement de cet algorithme. L'évaluation de règles récursives peut être infinie et dans ce cas, la vérification de la requête formulée ne pourra jamais se terminer. De plus, dans l'arbre de recherche, un sous but peut être dérivé plusieurs fois et son évaluation est donc répétée [Ull89].

Les autres problèmes liés à l'inférence par chaînage arrière sont discutés dans le livre [RN03] et ne sont pas plus développés dans ce travail.

3.3 Les ensembles magiques

Comme il a été expliqué précédemment, l'évaluation des règles selon l'algorithme 1 du chaînage avant (voir page 36) comporte des défauts. Cet algorithme est fortement ralenti par la découverte de faits ne participant pas à la vérification de la réalisation du but. Pour cette raison, les programmes de règles sont souvent évalués en chaînage arrière. Cette deuxième méthode fournit une réponse plus rapide à une requête et minimise le nombre d'évaluations répétées d'un même fait. Lors de l'exécution, la différence principale de ce processus par rapport au chaînage avant réside dans le fait qu'elle tient compte du but à rechercher. Cependant, la technique des *ensembles magiques* ("magic sets") transforme des règles de la forme de clauses de *Horn* pour les évaluer ensuite en chaînage avant.

Mais pourquoi vouloir à tout prix utiliser une telle méthode alors que celle du chaînage arrière est plus performante ? Simplement car certaines règles peuvent bloquer le processus d'évaluation en chaînage arrière. Les défauts de cette méthode sont brièvement abordés dans la section précédente.

La technique des *ensembles magiques* conserve les avantages du chaînage arrière en transformant les règles du programme. Celles-ci sont réécrites afin de fournir un nouvel ensemble de règles dans lesquelles le but à réaliser est intégré. Lors de l'exécution de l'algorithme de chaînage avant, le nombre de faits inférés est moins important car seuls les faits pertinents sont calculés.

Dans cette partie du travail, les différents concepts utiles à la compréhension de la méthode des *ensembles magiques* sont expliqués. Ensuite, le processus de transformation des règles est détaillé.

3.3.1 Les différents concepts

La *décoration* des prédicats

La description de la *décoration* ("adornment") des prédicats est tirée du livre [Ull89].

Soit le prédicat $p(t_1, t_2, ..., t_k)$. La *décoration* de ce prédicat consiste à lui attribuer une chaîne de caractères de longueur k contenant les symboles de *décoration* en tenant compte des faits de la base de connaissances. Ces symboles sont décrits avec l'alphabet $\{b, f\}$. Si le i^{ime} symbole de *décoration* de p est b (pour "bound") alors le i^{ime} argument de p est donné. Dans le cas contraire, le symbole associé au prédicat est f (pour "free") et il représente une variable libre. Cela signifie que cet argument sera instancié lors de l'évaluation de la règle dont le prédicat p fait partie.

La règle récursive permettant de trouver les ancêtres d'une personne illustre les propos tenus ci-dessus.

$$hasAncestor(X, Y) \leftarrow hasParent(X, Z) \wedge hasAncestor(Z, Y)$$

Soit $hasAncestor(john, Y)$, le but à fixer. L'information contenue dans ce but permet de spécifier la *décoration* du prédicat *hasAncestor*. Dans ce prédicat, la variable X est

donnée tandis que la variable Y doit encore être évaluée. De ce fait, le prédicat *décoré* est $hasAncestor^{bf}(X, Y)$. Les variables du prédicat *hasParent* seront instanciées après l'évaluation de cette règle. La *décoration* de celui-ci est $hasParent^{ff}(X, Z)$.

Les prédicats magiques

La transformation du prédicat p en un prédicat *magique* se fait en deux étapes. Tout d'abord, ce prédicat est *décoré*. Ensuite, il est renommé en m_p ("m" comme *magique*). Dès lors, il désigne un prédicat *magique*. Les arguments de m_p sont les arguments donnés du prédicat *décoré* p (les arguments auxquels le symbole b est associé). Les arguments spécifiés libres dans le prédicat de *décoration* sont ignorés.

En considérant le but $hasAncestor(john, Y)$, la transformation du prédicat $hasAncestor(X, Y)$ en un prédicat *magique* donne $m\_hasAncestor(X)$. Ce nouveau prédicat est vrai si et seulement s'il est évalué à $m\_hasAncestor(john)$. Les autres arguments du prédicat *hasAncestor* sont mis de côté car ils ne servent pas à la vérification du but.

3.3.2 La transformation de base en *ensembles magiques*

La méthode présentée dans cette partie du mémoire est la simplification de transformation en *ensembles magiques* décrite dans [BR91].

L'objectif de cette méthode est de définir des prédicats additionnels calculant la valeur passée d'un prédicat vers les autres dans les règles. Les règles originales sont modifiées afin d'être déclenchées seulement quand les valeurs sont disponibles pour les prédicats additionnels. Ces prédicats sont les prédicats *magiques* et les valeurs qu'ils calculent forment les ensembles *magiques*. La transformation des règles a pour conséquence de réduire l'espace de recherche lors de l'évaluation du but concerné.

La technique des ensembles *magiques* est une méthode automatique de transformation des règles et d'une requête. La réécriture de ces règles est correcte car leur évaluation en chaînage avant produit les mêmes réponses qu'avec l'ensemble de règles non transformées pour le but concerné. Ce processus de transformation prend en entrée les règles du programme et une requête de la forme $q(s_1, s_2, ..., s_n)$[1]. A la sortie, un nouvel ensemble de règles optimisé est proposé.

Le processus général de transformation est décrit ci-dessous.

1. Les règles du programme sont transformées en un ensemble de règles *décorées*. Pour chacune des règles originales, les prédicats sont remplacés par les prédicats *décorés*. Seules les règles participant à la réalisation du but sont transformées.
2. Pour chaque règle r faisant partie des règles *décorées*, et pour chaque occurrence d'un prédicat *décoré* p^a dans son corps, une règle *magique* est générée. Le prédicat de tête de cette nouvelle règle est le prédicat *magique* m_p^a associé au prédicat *décoré* p^a. Si le prédicat *décoré* de la tête de la règle r a au moins une variable donnée, alors le prédicat *magique* correspondant à ce prédicat est ajouté au corps de la nouvelle

[1] Dans le cas du langage *SWRL*, la requête est soit une description soit une propriété *SWRL*

règle. Les prédicats du corps de la règle r sont aussi ajoutés au corps de la règle *magique*.

3. Les règles avec les prédicats *décorés* sont modifiées. Pour chacune d'elles, le prédicat *magique* correspondant au prédicat *décoré* de la tête est ajouté au corps de celle-ci.
4. La source des prédicats *magique*s est créée à partir du but à réaliser. Celui-ci fournit la valeur initiale pour le prédicat *magique* correspondant à la requête.

Les règles *magiques* calculent toutes les associations atteignables à partir de la source (qui reflète l'objectif de base). Ces ensembles d'associations sont appelés les ensembles magiques.

Les règles et le but considérés servant d'exemples pour illustrer la transformation en ensembles magiques sont reprises ci-dessous.

Le but à réaliser est $hasAncestor(john, Y)$.

Les règles utilisées sont détaillées ci-dessous.

$$hasAncestor(X, Y) \leftarrow hasParent(X, Y)$$

$$hasAncestor(X, Y) \leftarrow hasParent(X, Z) \wedge hasAncestor(Z, Y)$$

La transformation de ces règles en règles *décorées* est :

$$hasAncestor^{bf}(X, Y) \leftarrow hasParent(X, Y)$$

$$hasAncestor^{bf}(X, Y) \leftarrow hasParent(X, Z) \wedge hasPncestor^{bf}(Z, Y)$$

Dans la deuxième règle *décorée*, il y a un prédicat *décoré* $hasAncestor^{bf}(Z, Y)$. De ce fait, la règle *magique* reprise ci-dessous est créée.

$$m\_hasAncestor^{bf}(Z) \leftarrow m\_hasAncestor^{bf}(X) \wedge hasParent(X, Z) \tag{3.1}$$

La troisième étape du processus constiste à modifier les règles *décorées*.

$$hasAncestor^{bf}(X, Y) \leftarrow m\_hasAncestor^{bf}(X) \wedge hasParent(X, Z) \tag{3.2}$$

$$\begin{aligned} hasAncestor^{bf}(X, Y) \quad \leftarrow \quad & m\_hasAncestor^{bf}(X) \\ & \wedge hasParent(X, Z) \\ & \wedge hasAncestor^{bf}(Z, Y) \end{aligned} \tag{3.3}$$

Et pour terminer ce processus, la source $m\_hasAncestor^{bf}(john)$ est générée.

Le lecteur désirant plus d'informations concernant la transformations en *ensembles magiques* peut consulter [BR91]

3.4 L'algorithme *Rete*

Les idées à la base de cette partie de ce mémoire sont tirées de [Doo95] excepté lorsqu'une autre source est citée.

Il a été vu précédemment que l'exécution de l'algorithme 1 de chaînage avant (voir page 36) sur un programme de règles n'est pas optimal car certains faits sont évalués plusieurs fois. L'algorithme *Rete*, publié dans [For82] en 1982, permet d'éviter cette évaluation répétée en construisant un réseau de noeuds. Ce réseau a pour but de structurer l'information contenue dans le programme à traiter. Lors de l'ajout, de la suppression ou de la modification de faits, toutes les règles ne doivent pas être réexaminées. L'analyse du réseau permet d'identifier les éléments à réévaluer. L'algorithme *Rete* diminue donc le temps nécessaire à l'évaluation du programme. Cependant, cette optimisation est faite au détriment de la quantité de mémoire utilisée car toutes correspondances entre les conditions et les faits sont mémorisés.

Dans cette partie du mémoire, les deux parties du réseau nécessaires à l'exécution de l'algorithme *Rete* sont décrites avant de s'attarder sur le fonctionnement de l'algorithme.

3.4.1 Le stockage de l'information

Pour éviter l'évaluation répétée des clauses d'une règle, l'algorithme *Rete* mémorise l'information qu'il utilise et qu'il découvre. Elle est stockée soit dans la *mémoire de travail* ("working memory"), soit dans la *mémoire de production* ("production memory"). La *mémoire de travail* contient les faits décrivant le domaine à un moment particulier. Ces faits sont en fait des triples[2] composés d'un identifiant, d'un attribut et d'une valeur. Ces triples ont la même structure que les triples *RDF* et ne contiennent que des constantes. A chaque fois qu'un nouveau fait est découvert, il est ajouté à la *mémoire de travail*. Chaque élément de la *mémoire de travail* est appellé *WME* (l'abréviation de "working memory element"). La *production memory* regroupe l'ensemble des règles du programme. Les clauses du corps de la règle sont les conditions tandis que les clauses de la tête sont les actions. La *mémoire de travail* et la *mémoire de production* servent de base à l'algorithme *Rete*. A chaque fois qu'elles changent, l'algorithme est réexécuté.

Afin d'illustrer les concepts décrits dans cette partie du travail, les règles suivantes sont prises en exemple :

$$P1 : \ hasDaughter(X, Y) \leftarrow hasChild(X, Y) \wedge Woman(Y)$$

$$P2 : \ fatherOfAlice(X) \leftarrow hasParent(X, alice) \wedge Man(X)$$

$$P3 : \ hasAunt(X, Y) \leftarrow hasParent(X, alice) \wedge hasSister(Z, Y)$$

Les tableaux 3.3 (voir page 45) décrivent les *conditions de production* et la *mémoire de travail* du début de l'exécution associés à cet exemple.

[2]En réalité, l'algorithme *Rete* est exécuté à partir de tuples. Dans ce mémoire, les concepts sont simplifiés. Les tuples sont considérés comme étant des triples

	Les CDP[1]
C1	$hasParent(X, alice)$
C2	Man(X)
C3	$hasSister(X, Y)$
C4	$hasChild(X, Y)$
C5	Woman(Y)

	La mémoire de travail
W1	hasParent(bob,alice)
W2	hasParent(john,chris)
W3	hasSister(alice,ema)
W4	Man(chris)
W5	Man(bob)
W6	Woman(alice)
W7	hasChild(alice,bob)

TAB. 3.3 – Les conditions de production et la mémoire de travail associés à l'exemple

3.4.2 Le réseau *alpha*

Le réseau *alpha* est, comme la figure 3.4 (voir 48) le montre, la partie droite du réseau construit par l'algorithme *Rete*. Celle-ci permet de trier les faits de la *mémoire de travail* en les filtrant par motif. Le réseau *alpha* peut être représenté par un arbre dont chacun des noeuds est une partie d'une condition reprise dans le corps d'une règle de la *mémoire de production*. L'arbre comporte autant de feuilles qu'il n'y a de conditions différentes dans les corps des règles. Les feuilles de l'arbre constituent les zones de mémoire *alpha*. Chacune d'elles contient l'ensemble des faits s'unifiant avec les conditions présentées par les noeuds parcourus dans le réseau *alpha*. Le chemin de la racine de l'arbre à cette mémoire reprend tous les éléments d'une condition donnée. La racine de cet arbre est neutre car elle n'est pas testée (elle s'unifie avec tous les faits de la *mémoire de travail*) [Alb89].

Chaque élément de *la mémoire de travail* passe à travers cet arbre et est placé dans la mémoire *alpha* adéquate. Les formalismes de représentation utilisés pour illustrer cet exemples sont tirés de [Doo95].

Le réseau *alpha* associé à la *mémoire de travail* présentée précédemment, est repris dans la figure 3.2 (voir page 46).

[1] Les conditions de production

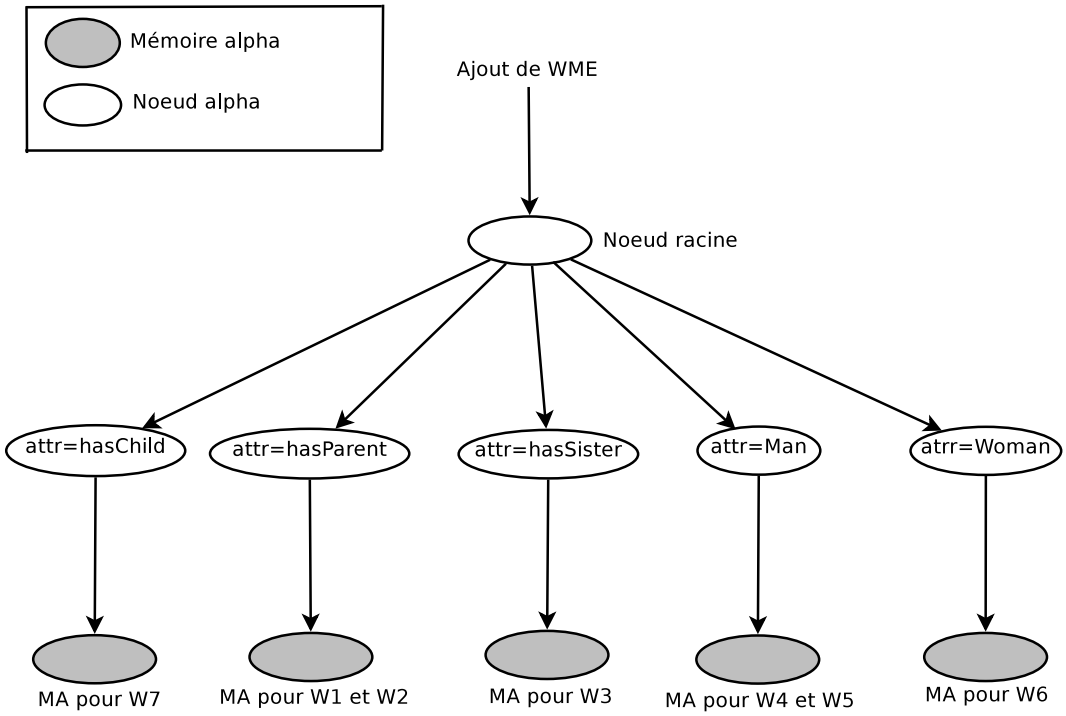

FIG. 3.2 – Le réseau *alpha*

Il est possible de représenter le réseau *alpha* sous différentes structures. Le lecteur intéressé peut trouver plus d'informations dans [Doo95].

3.4.3 Le réseau *beta*

Le réseau *beta* est composé des noeuds de *jonction*, des noeuds de *production* et des noeuds contenant les zones de mémoire *beta*.

Le réseau *beta* gère les correspondances entre les éléments du réseau *alpha* et la *mémoire de production*. Chaque condition apparaissant dans le corps d'une règle forme un noeud dans le réseau *beta*. Le successeur de ce noeud contient la condition suivante dans le corps de cette règle. Si un noeud à plusieurs successeurs, cela signifie que la condition de ce noeud fait partie de plusieurs corps de règles distinctes. Un noeud terminal de ce réseau est un noeud de *production* (un *p-node*).

Chaque noeud de *jonction* est le point de rencontre entre un élément de la mémoire *alpha* et une zone mémoire *beta*. Ce noeud détermine les éléments de la mémoire *alpha* qui respectent les conditions reprises dans la mémoire *beta*. Soit un noeud de *jonction* particulier dont les entrées sont une zone mémoire *alpha* et une zone mémoire *beta*. Certains *WME* de la mémoire *alpha* satisfont les condition du noeud *beta*. Dès lors, la combinaison de ces *WME* avec les *WME* des noeuds *beta* précédant le noeud de *jonction* forment les instanciations partielles des productions. Celles-ci sont appelées des *jetons* et sont stockées dans la mémoire *beta*. Chaque *jeton* créé est passé au noeud *beta* suivant dans le réseau.

Lorsque un *jeton* atteint un *p-node*, cela signifie que les instanciations qu'il représente satisfont toutes les conditions du corps d'une règle équivalent au différents noeuds traversé par ce *jeton* dans le réseau *beta*.

Le réseau *beta* associé à la *mémoire de production* présentée précédemment, est repris dans la figure 3.3 (voir page 47). Les formalismes de représentation utilisés pour illustrer cet exemples sont tirés de [Doo95].

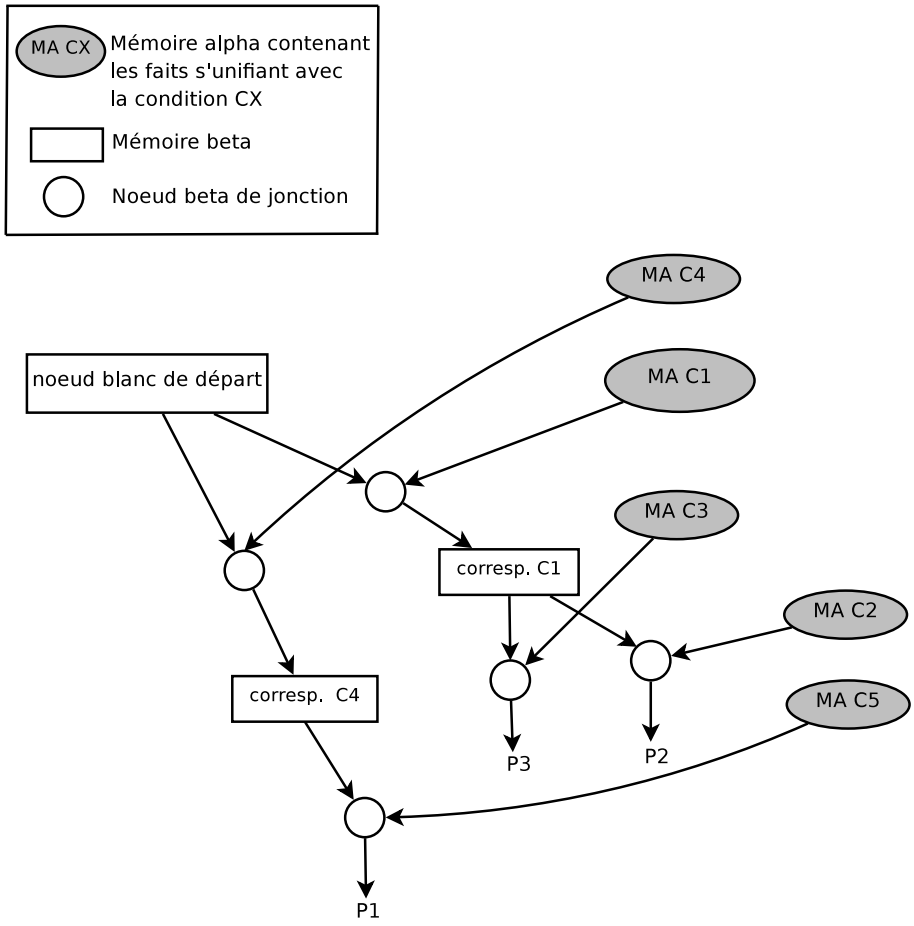

FIG. 3.3 – Le réseau *beta*

3.4.4 L'algorithme

Lorsqu'un fait est ajouté à la *mémoire de travail*, il est placé à l'entrée du réseau *alpha* qui est activé. L'activation de ce réseau a pour objectif de placer ce nouveau fait dans la zone mémoire *alpha* correspondant à son motif. A chaque noeud de l'arbre, le fait est redirigé vers le successeur respectant sa composition. Lorsque le fait est enregistré dans la mémoire *alpha*, le noeud de *jonction* analyse la situation et détermine si le fait ajouté satisfait la condition du noeud *beta*. Si c'est le cas, un nouveau *jeton* est créé et le réseau *beta* est activé. Ce *jeton* est passé au noeud *beta* suivant et l'activation du réseau *beta* se fait de proche en proche. Quand le *jeton* atteint le fond du réseau, cela signifie que

l'instanciation satisfait toutes les conditions d'une règle. Dans ce cas, l'action de la tête de cette règle est évaluée.

Il est utile de remarquer que l'algorithme *Rete* gère l'ajout et la suppression d'éléments de la *mémoire de travail* ainsi que l'ajout et la suppression de production (voir [Doo95]).

Le réseau de la figure 3.4 reprend les réseaux *alpha* et *beta* identifiés précédemment (figures 3.2 et 3.3 respectivement). L'état de ce réseau est obtenu après avoir déclenché toutes les activations possibles dans le réseau *alpha* et dans le réseau *beta*. Les *p-nodes* $p2$ et $p3$ sont atteints et les faits $hasDaughter(bob, ema)$, $fatherOfAlice(bob)$. Les formalismes de représentation utilisés pour illustrer cet exemple sont tirés de [Doo95].

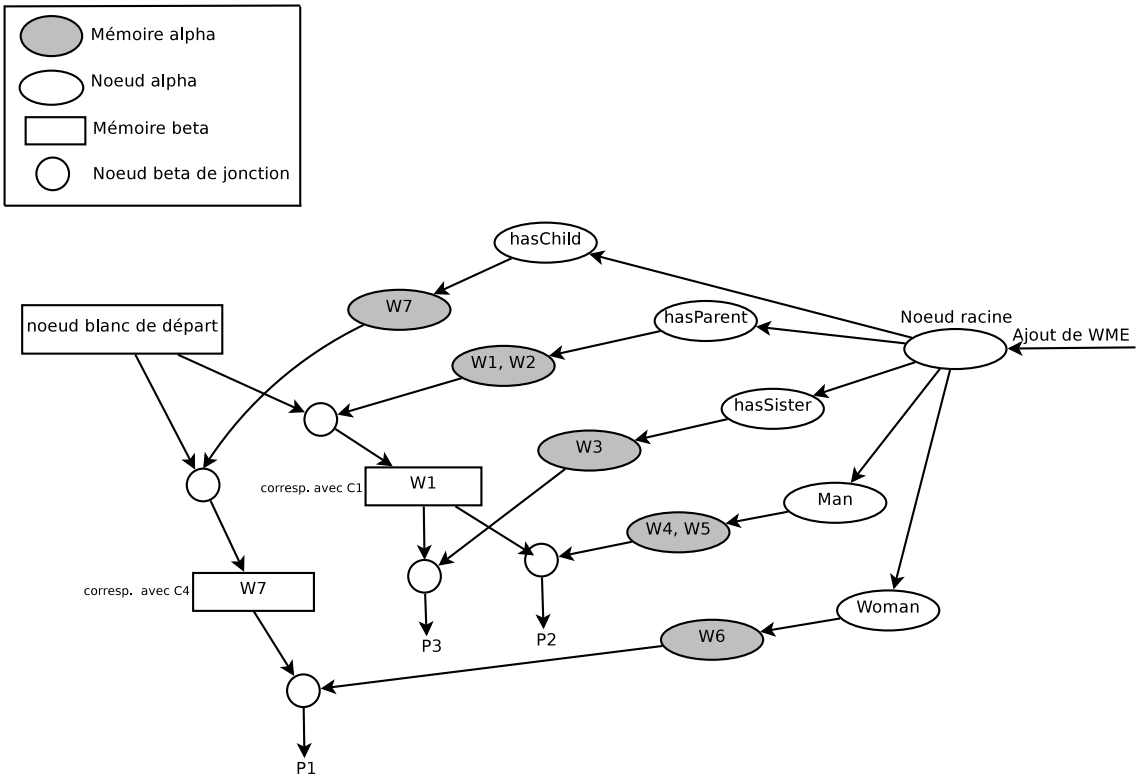

FIG. 3.4 – Le réseau de noeuds construit par l'algorithme *Rete*

3.4.5 L'apport de l'algorithme *Rete*

L'algorithme *Rete* rend l'évaluation d'un programme de règles plus rapide grâce à deux optimisations :

Le partage des noeuds : le partage des noeuds entre les différentes productions possédant les mêmes conditions évite les évaluations répétées.

L'enregistrement des états : Chaque changement réalisé soit dans la *mémoire de travail* ou soit dans la *mémoire de production* est propagé à travers le réseau *alpha* et le

réseau *beta*. Toute l'information découverte durant les exécutions précédentes de l'algorithme sont toujours disponibles dans chacun de ces réseaux. De ce fait, il n'est pas nécessaire de réévaluer toutes les connaissances. Seuls les noeuds ayant été modifiés doivent être réévalués. Cependant, toutes les modifications doivent être propagées à travers les deux réseaux.

Chapitre 4

L'évaluation en chaînage avant d'un ensemble de règles *SWRL*

Comme il a été spécifié dans l'introduction, l'objectif de ce mémoire est de présenter le moteur d'inférences créé à l'entreprise *Mission Critical*. Celui-ci est capable d'évaluer un ensemble de règles de type *SWRL* en chaînage avant. Le langage *SWRL* et l'inférence qu'il est possible de faire sur les règles décrites par ce langage ont été exposés dans les parties 2.3 et 3.1 de ce travail.

L'implémentation du moteur d'inférences en chaînage avant détaillé dans ce chapître, s'inspire de l'algorithme 1 (voir page 36) dit "naïf". Le problème de cet algorithme est qu'il ne tient pas compte du fait que les règles *SWRL* font référence à des concepts définis dans des ontologies. Une méthode correcte se doit d'évaluer ces règles en appelant le raisonneur de la logique de description. Le moteur d'inférences proposé dans ce mémoire va plus loin que l'algorithme "naïf" car il évalue les règles en considérant les spécificités du langage *SWRL*. De plus, il effectue des optimisations lors de l'évaluation de ces règles. L'implémentation du moteur d'inférences n'est pas donnée dans cet ouvrage car l'accent est mis sur la méthodologie et non pas sur un code écrit dans un langage de programmation particulier.

A travers ce chapître, le lecteur est en mesure de comprendre comment un ensemble de règles *SWRL* est évalué en chaînage avant. Cette évaluation se fait en deux étapes. La première consiste à déterminer un ordre de traitement efficace de ces règles avant de s'attarder à l'évaluation des éléments qui composent ces règles. Ensuite, la critique du moteur de règles, mettant en avant ses avantages et ses défauts, est réalisée. Et pour terminer cette partie, des améliorations possibles visant à atteindre une méthode plus efficace sont expliquées. Celles-ci s'inspirent des points théoriques concernant la transformation en *ensembles magiques* et l'algorithme *Rete* afin de fournir une méthode d'évaluation plus efficace.

4.1 Première étape : L'analyse des relations entre les règles

L'algorithme 1 de chaînage avant (voir page 36) s'exécute sur un ensemble de règles. Toutes les règles de cet ensemble sont évaluées l'une après l'autre en fonction de leur ordre d'apparition dans le fichier *SWRL*. Mais avec cette technique, l'évaluation de certaines règles peut être incomplète car elle nécessite l'inférence préalable d'autres faits issus d'autres règles. Il existe une dépendance entre quelques une de ces règles dont il faut tenir compte lors de l'évaluation de l'ensemble de règles. Le langage *SWRL* n'imposant pas d'ordre de déclaration des règles, il y a peu de chance que celles-ci aient été placées dans un ordre respectant ces dépendances. De ce fait, cette technique risque fort d'être inefficace et le nombre d'inférences réalisées sur l'ensemble des règles non optimal. Ce nombre peut être augmenté à condition d'examiner les règles selon un ordre déterminé.

L'ordre d'évaluation des règles doit donc faire l'objet d'une étude afin de déterminer quelles sont les règles à traiter avant les autres. Une règle ne sera évaluée que lorsque toutes les règles dont elle dépend auront déjà été traitées. L'analyse des dépendances fait partie d'un processus visant à mener à bien l'évaluation de l'ensemble des règles apparaissant dans le programme *SWRL*. Ce processus comprend les étapes reprises ci-dessous qui doivent être exécutées en suivant l'ordre d'énonciation.

1. La construction d'un graphe de dépendances entre les règles ;
2. L'exécution de l'algorithme de Tarjan identifiant les composantes fortement connexes d'un graphe ;
3. L'agrégation des noeuds qui définissent un circuit ;
4. L'exécution du tri topologique entre les noeuds du graphe ;
5. L'évaluation ordonnée des règles selon le tri.

L'ensemble des règles reprises ci-dessous est utilisé afin d'illustrer les propos tenus dans cette partie de ce travail.

Règle 4.1.1 ***Def-hasParent***

$$hasParent(?Child, ?Parent) \leftarrow hasChild(?Parent, ?Child)$$

Règle 4.1.2 ***Def-hasChild***

$$hasChild(?Parent, ?Child) \leftarrow hasParent(?Child, ?Parent)$$

Règle 4.1.3 ***Def-hasUncle***

$$hasUncle(?Child, ?Uncle) \leftarrow hasChild(?Parent, ?Child) \wedge hasBrother(?Parent, ?Uncle)$$

Règle 4.1.4 ***Def-hasDirectAncestor***

$$hasAncestor(?Parent, ?Ancestor) \leftarrow hasParent(?Parent, ?Ancestor)$$

Règle 4.1.5 ***Def-hasAncestors***

$$hasAncestor(?Child, ?Ancestor) \leftarrow hasParent(?Child, ?Parent) \wedge hasAncestor(?Parent, ?Ancestor)$$

Règle 4.1.6 ***Def-hasGrandChild***

$$hasGrandChild(?Father, ?GrandChild) \leftarrow hasChild(?Parent, ?GrandChild) \wedge hasParent(?Father, ?GrandFather) \wedge Male(?Father) \wedge sameAs(?Parent, ?Father)$$

Règle 4.1.7 ***Def-olderThan***

$$swrlb : lessThan(?AgeChild, ?AgeParent)^{1} \leftarrow hasParent(?Child, ?Parent)) \wedge hasAge(?Father, ?AgeFather) \wedge hasAge(?AgeChild)$$

L'évaluation de la règle 4.1.7 ne permet pas d'ajouter de nouveaux faits à la base de connaissances. Son rôle est de vérifier la cohérence du modèle. En effet, tout parent doit être plus âgé que son enfant.

4.1.1 Le graphe des dépendances des règles

La construction de cette structure de données est utile afin d'analyser les dépendances entre les règles *SWRL*. Le graphe de dépendances des règles est constitué d'arcs et de noeuds. La notation mathématique de ce graphe orienté est G=(X,U) où X est l'ensemble des noeuds et U est l'ensemble des arcs.

Un noeud contient l'identifiant d'une règle. Un arc relie deux noeuds si un des prédicats du conséquent de la règle représentée par le noeud de départ apparait dans l'antécédent de la règle dont l'identifiant est dans le noeud d'arrivée. Le nom de l'arc est le prédicat commun aux deux règles des noeuds. Un arc est construit entre deux noeuds pour exprimer le fait que l'évaluation de la règle reprise dans le noeud d'arrivé pointé par cet arc nécessite la découverte de nouveaux faits obtenus par l'évaluation de la règle reprise dans le noeud de départ de cet arc. Les dépendances entre les règles sont générées par les prédicats se rapportant aux concepts d'une ontologie (un prédicat correspondant à une des deux premières lignes du tableau 2.2 (voir page 24)). Les atomes prédéfinis par le langage *SWRL* (représentés par les autres lignes du tableau 2.2) ne peuvent pas générer de dépendances avec d'autres règles car leur évaluation ne permet pas de découvrir de nouveaux faits à ajouter à la base de connaissances.

La structure de graphe permet d'analyser les dépendances entre les règles en appliquant systématiquement des algorithmes de le théorie des graphes.

[1] Ce built_in impose que l'âge d'un enfant soit inférieur à celui de son parent

Après l'analyse des deux règles 4.1.2 et 4.1.3 énoncées précédemment, deux noeuds sont crées. Le premier contient *Def-hasChild* et le second *Def-hasUncle*. Il existe une dépendance entre ces noeuds qui sont dès lors reliés par un arc dont le nom est *hasChild*. En effet, le prédicat *hasChild* présent dans le conséquent de la règle 4.1.2 se retrouve dans l'antécédent de la règle 4.1.3. Cette dépendance signifie qu'avant de pouvoir évaluer la règle 4.1.3 il est nécessaire d'évaluer la règle 4.1.2.

L'analyse des dépendances de toutes les règles déclarées à la page 52 produit un graphe illustré par la figure 4.1, page 54.

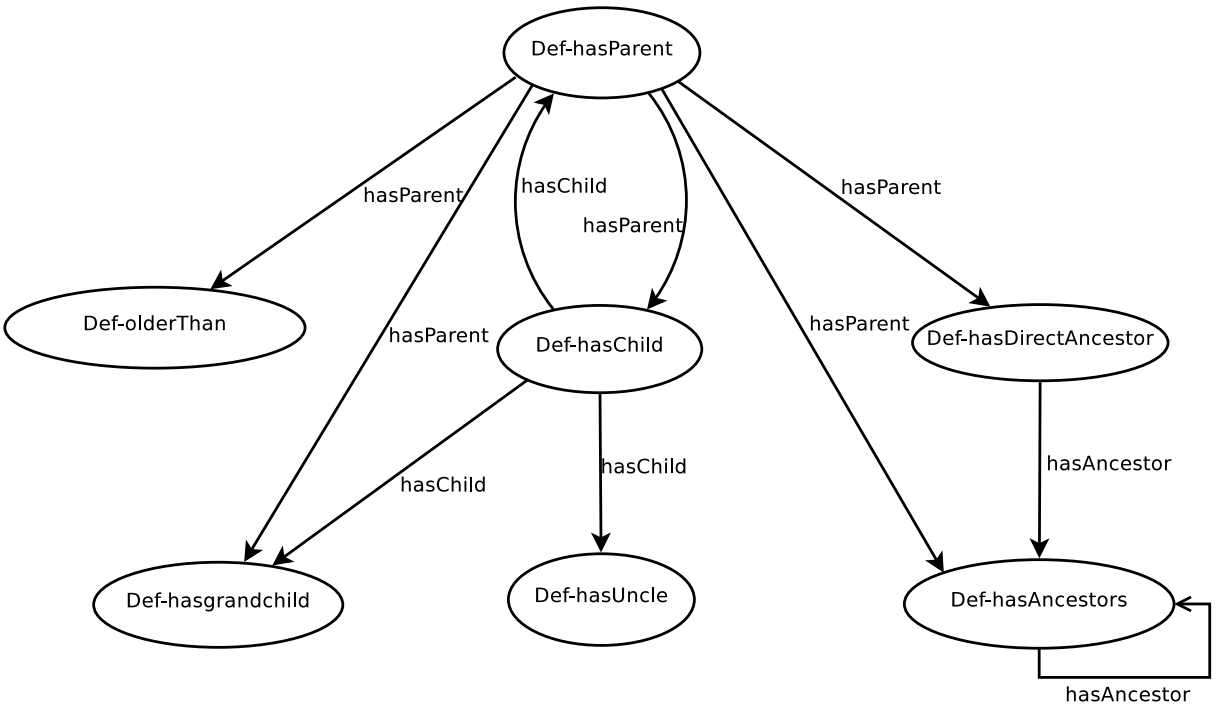

FIG. 4.1 – Le graphe des dépendances entre les règles

4.1.2 L'algorithme de Tarjan

L'algorithme de Tarjan permet d'identifier les composantes fortement connexes d'un graphe. Ces composantes sont utiles afin d'identifier les noeuds faisant partie d'un circuit dans ce graphe.

Avant de continuer l'explication de cette section, il est utile de rappeler quelques définitions propres à la théorie des graphes.

Définition 4.1.1 *Un chemin est une suite d'arcs*

$$\mu = (u_1, u_2, , u_i, u_{i+1}, , u_q)$$

telle que l'extrémité terminale d'un arc $u_i (i = 1, ..., q-1)$ *coïncide avec l'extrémité initiale de l'arc suivant* u_{i+1}.

Définition 4.1.2 *Une composante fortement connexe est un sous-ensemble de sommets X du graphe G=(X,U) tel qu'il existe un chemin entre deux sommets quelconques de la composante.*

Cette algorithme fait une recherche en profondeur d'abord des composantes fortement connexes du graphe orienté. Il s'exécute au départ d'un noeud et permet d'identifier les composantes fortement connexes dans le sous-graphe atteignable à partir de ce noeud. Après la visite d'un noeud, la méthode évalue ses noeuds suivants. Si l'évaluation des suivants conduit au noeud de départ, c'est que l'ensemble de ces noeuds constitue une composante fortement connexe. L'idéal est d'exécuter ce calcul sur tous les noeuds du graphe afin de trouver toutes les composantes fortement connexes car il est impossible d'identifier les composantes faisant partie des autres sous-graphe inatteignables à partir de ce noeud. Le défaut de cette méthode est qu'elle est loin d'être optimale car certains noeuds vont être visités plusieurs fois. Il serait donc intéressant de retenir les noeuds déjà traités.

Après la visite de tous les noeuds du graphe, les résultats sont combinées afin de déterminer les composantes connexes les plus grandes possibles.

L'algorithme de Tarjan appliqué au graphe de la figure 4.1, page 54 détecte les composantes fortement connexes suivantes :
- *Def-hasAncestors*
- L'ensemble composé de *Def-hasParent* et *Def-hasChild*

4.1.3 L'agrégation des noeuds qui définissent un circuit

L'étape suivante du processus consiste à agréger les noeuds du graphe de dépendances qui forment un circuit afin d'être en mesure de déterminer l'ordre d'évaluation des règles. Ces circuits sont les composantes fortement connexes identifiées à l'étape précédente de ce processus. Sans l'agrégation des noeuds définissant un circuit, l'algorithme de tri topologique ne peut pas déterminer d'ordonnancement basé sur la précédences pour ces noeuds.

Un nouveau graphe de noeuds agrégés est donc construit. Dans cette structure, un noeud est soit un noeud simple représentant une règle soit un noeud agrégé qui est l'union de plusieurs règles. Les noeuds identifiés par l'algorithme de Tarjan comme des noeuds faisant partie d'un circuit deviennent des noeuds agrégés et les autres noeuds sont transformés en noeuds simples.

Les relations entre les noeuds du graphe de noeuds agrégés sont construites en fonction de la transformation opérée sur chaque paire de noeuds reliés par un arc dans le graphe de départ. Si ces noeuds sont devenus :

Deux noeuds simples : Un nouvel arc avec le même nom que l'arc de départ est ajouté entre ces noeuds du nouveau graphe ;

Un noeud simple et une partie d'un noeud agrégé : Un arc est créé dont le nom est le prédicat commun à la règle représentée par le noeud simple et une des règles de l'agrégation.

Deux parties de deux agrégations différentes : Le nom de l'arc ajouté qui relie ces deux composantes est le prédicat commun aux deux règles présentes dans les deux parties de l'agrégation ;

Deux parties de la même agrégation : Il n'y a plus d'arc à créer car ces deux noeuds n'en forment désormais plus qu'un seul.

Un noeud qui bouclait sur lui-même empêche la construction de l'ordre non stricte car il fait partie de ses suivants. Il devient lui aussi un noeud agrégé.

La transformation du graphe de la figure 4.1 page 54 en graphe de noeuds agrégés, est représentée à la figure 4.2 page 56.

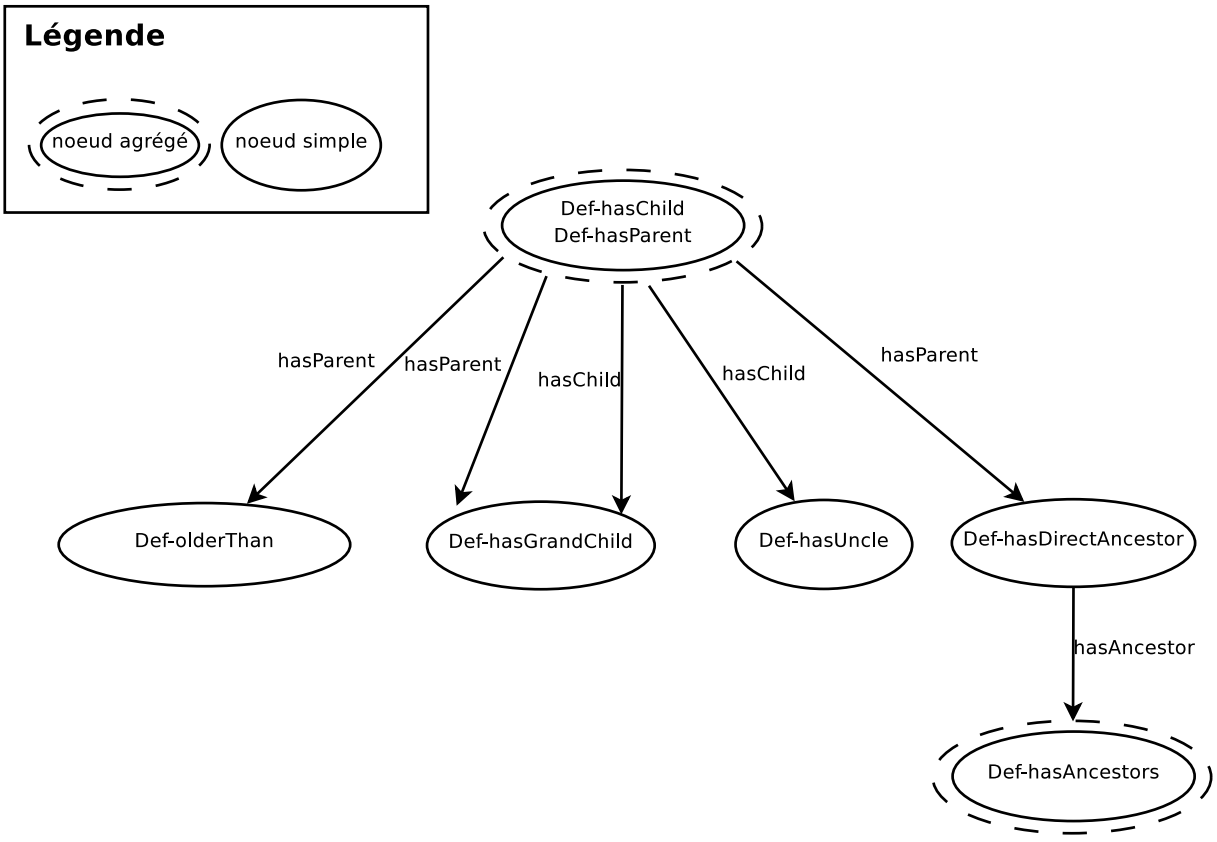

FIG. 4.2 – Le graphe des noeuds agrégés

4.1.4 Le tri topologique

Le tri topologique d'un graphe sans circuit est la détermination d'un ordre partiel de l'ensemble de ses noeuds en fonction du nombre de leurs suivants. Cette ordonnancement revient à attribuer un rang à chaque noeud du graphe.

Dans cet ordonnancement,

1. Un noeud a un rang inférieur à tous ses suivants ;
2. Un noeud a un rang inférieur à un autre si le nombre de noeuds suivants ce noeud est plus important que l'autre noeud ;
3. Il n'existe pas de chemin reliant des noeuds de même rang ;
4. Les noeuds de rang 1 n'ont pas de précédents tandis que les noeuds de rang le plus élevé ne possèdent pas de suivant.

Cet algorithme parcourt les noeuds en profondeur d'abord et ne peut s'exécuter sur un graphe contenant des circuits. A partir d'un noeud, la méthode visite tous les suivants afin de leur attribuer un rang supérieur. Si un noeud fait partie d'un circuit, il est impossible de lui attribuer un rang inférieur à ses noeuds suivants car il est aussi leur suivant. C'est pourquoi cet algorithme est exécuté sur le graphe contenant les noeuds agrégés obtenus à l'étape précédente de ce processus. Cette technique garantit le respect des dépendances entre les règles lors de l'évaluation de celles-ci.

La complexité de cette algorithme est de $\theta(n + m)$ où n est le nombre de noeuds du graphe et m le nombre d'arcs.

Le tri topologique de la figure 4.2 page 56 trouve l'ordre d'évaluation suivant :

1. *Def-hasChild, Def-hasParent*
2. *Def-OlderThan*
3. *Def-hasGrandChild*
4. *Def-hasUncle*
5. *Def-hasDirectAncestor*
6. *Def-hasAncestors*

Toutes les permutations des règles 2, 3, 4, 5 sont possibles car ces noeuds ont le même rang.

4.1.5 L'évaluation ordonnée : gestion des circuits

L'évaluation des règles qui forment un circuit se fait par la méthode du point fixe. Le langage *SWRL* est un langage monotone ce qui signifie qu'il est seulement possible d'ajouter des faits à la base des connaissances et non pas d'en retirer. Lorsqu'un ensemble de règles est évaluée plusieurs fois, le nombre de faits inférés est soit supérieur ou reste constant par rapport à l'évaluation précédente. Un point fixe est atteint si le nombre d'inférences pour un ensemble de règles est constant d'une évaluation à l'autre de celles-ci.

Lors de l'évaluation d'un ensemble de règles constituant un circuit, le nombre d'inférences réalisées à chaque évaluation est retenu. Si ce nombre est identique lors de deux évaluations successives, cela signifie que le point fixe est atteint et qu'aucun nouveau fait ne peut être déduit. Sinon l'évaluation est poursuivie car de nouveaux faits peuvent encore être inférés

Toutes les règles du circuit sont évaluées séparément par la méthode d'évaluation des noeuds simples décrite ci-après.

4.1.6 L'évaluation des noeuds simples

Les noeuds simples représentent une seule règle *SWRL*. Celle-ci est traitée de manière isolée selon le principe d'évaluation décrit dans la section suivante.

4.2 Deuxième étape : L'évaluation d'une règle *SWRL*

Comme il a été vu précédemment, une règle *SWRL* est composée d'un antécédent (un corps) et d'un conséquent (la tête) contenant tous deux une conjonction d'atomes. L'évaluation d'une règle *SWRL* est un processus permettant de déterminer les différentes instanciations possibles des atomes du conséquent en fonction des substitutions des variables découvertes lors de l'évaluation des atomes de l'antécédent. Les différents types d'atomes apparaissant dans une règle *SWRL* sont repris dans le tableau 2.2 (voir page 24). Dans la suite de ce mémoire, le terme *dl_predicate* est utilisé pour désigner des atomes définissant des concepts *OWL*.

Avant de présenter le processus d'évaluation d'une règle *SWRL*, il est utile de préciser que le traitement des *dl_predicate*, des *built_in* et des *differentFrom* se fait par l'intermédiaire de raisonneurs différents. Les *dl_predicate* sont évalués par le raisonneur de la logique de description (le *dl_reasoner*) tandis que les autres types d'atomes sont traités par le raisonneur *SWRL*.

L'évaluation d'une règle se fait en 7 étapes décrites plus en détails dans la suite du rapport.

1. L'élimination des prédicats de type *sameAs*
2. La construction des composantes d'atomes liés sur base des variables
3. La construction du graphe de dépendances entre les composantes d'une règle
4. La détermination de l'ordre d'évaluation des composantes
5. L'évaluation des composantes selon leur ordre
6. L'évaluation du conséquent
7. La mise à jour de la base des faits

Les points 3 et 4 de ce processus ont pour but de créer une structure de données facilitant l'ordonnancement des atomes de l'antécédent et la sélection des substitutions des variables de ces atomes indispensable à l'évaluation du conséquent.

Lors de l'évaluation d'une règle, il est nécessaire de déterminer un ordre de traitement des atomes du conséquent sans quoi, le raisonneur risque d'effectuer du travail inutile. En effet, il détermine toutes les substitutions possibles des variables composant les atomes. Certaines d'entre elles ne sont pas pertinentes et sont éliminées lors de l'évaluation des autres atomes de la règle. Si l'ordre d'évaluation des atomes avait été inversé, ces substitutions superflues n'auraient sans doute pas été trouvées. Pour illustrer ces propos, la règle ci-dessous est prise en exemple.

Règle 4.2.1 ***Def-Adult***

$$\begin{aligned} Adult(?Person) \leftarrow\ & swrlb : greaterThanOrEqual(?Age, 18) \\ & \wedge\ hasAge(?Person, ?Age) \\ & \wedge\ Person(?Person) \end{aligned}$$

L'approche naïve consiste à traiter les atomes dans leur ordre d'apparition dans l'antécédent de la règle. Cette méthode évalue d'abord le *built-in greaterThanOrEqual*. Le résultat obtenu est l'ensemble de tous les entiers plus grands que 18. Cette énumération apporte très peu d'informations pertinentes car l'important dans cette règle est de trouver les individus de l'ontologie dont l'âge est supérieur à 18 ans. Lorsque le *dl_ reasoner* évalue les autres prédicats de cette règle, il élimine toutes les substitutions de la variable *Age* ne se rapportant pas à l'âge d'une personne de l'ontologie. Si par contre, les *dl_ predicate* de cette règle avaient d'abord été évalués, les substitutions de la variable *Age* correspondraient à l'âge des personnes de la base de connaissances. Ensuite, l'évaluation du *built_ in* aurait pour conséquence d'éliminer les personnes n'ayant pas 18 ans.

Il faut donc déterminer un ordre de traitement des atomes composant l'antécédent des règles. Il y a plusieurs possibilités d'ordonnancement. L'ordre choisi est sans conséquence sur le résultat obtenu car le langage *SWRL* fait partie des langages dits déclaratifs. Cependant, cet ordre minimise le nombre de substitutions non-pertinentes trouvées lors de l'évaluation des atomes de l'antécédent. Les prédicats de type *built_ in* et *differentFrom* sont évalués le plus tard possible dans le processus pour les raisons citées précédemment. Les *dl_ predicate* quant à eux sont traités en premier lieu.

L'exemple ci-dessus laisse aussi pressentir que les substitutions de certaines variables présentes dans l'antécédent d'une règle *SWRL* ne doivent pas être retenues. En effet, les substitutions de la variable *Age* ne doivent pas être prises en compte pour déterminer les instanciations de l'atome *Adult* de la tête. Ces substitutions ont une portée locale à l'antécédent.

Dans une règle, certaines substitutions n'apportent pas d'informations utilisées pour l'évaluation d'autres atomes et ne doivent donc pas être gardées. Il est intéressant de pouvoir identifier ces substitutions afin d'optimiser la méthode d'évaluation. Ne pas en tenir compte évite les calculs d'unification de ces substitutions avec les substitutions trouvées lors de l'évaluation des autres atomes de la règle.

Les points du processus permettant d'atteindre ces objectifs sont décrits plus en détails dans les sections suivantes de ce mémoire.

4.2.1 Les atomes *sameAs*

Les prédicats de type *sameAs* représentent des unifications entre deux termes. Il est possible de les effectuer statiquement au début du processus. Pour chacun des atomes de type *sameAs* d'une règle, les occurrences d'une des variables de l'équivalence est remplacée par l'autre dans tous les autres atomes. Après cette manipulation, les atomes de type *sameAs* peuvent être supprimés du corps de la règle car leur sémantique est impliquée par la version modifiée de la règle. La transitivité de la relation *sameAs* justifie la simplification énoncée ci-dessous. Si plusieurs atomes de type *sameAs* ont des variables communes dans leurs arguments, une de ces variables est choisie afin de remplacer toutes les occurrences des variables de ces atomes dans tous les autres atomes de la règle, y compris dans la tête.

La présence d'un atome de type *sameAs* dans la tête de la règle est traitée dans le

point concernant l'évaluation du conséquent de ce processus.

La règle 4.1.6 (voir page 53), reprise ci-dessous, est utilisée afin d'illustrer la transformation éliminant les atomes de type *sameAs*.

$$\begin{aligned} hasGrandChild(?Father, ?GrandChild) \leftarrow\ & hasChild(?Parent, ?GrandChild) \\ & \wedge hasParent(?Father, ?GrandFather) \\ & \wedge Male(?Father) \\ & \wedge sameAs(?Parent, ?Father) \end{aligned}$$

Après l'élimination de l'atome de type *sameAs*, la règle obtenue est décrite comme suit :

$$\begin{aligned} hasGrandChild(?Parent, ?GrandChild) \leftarrow\ & hasChild(?Parent, ?GrandChild) \\ & \wedge hasParent(?Parent, ?GrandFather) \\ & \wedge Male(?Parent) \end{aligned}$$

4.2.2 La construction des composantes d'atomes liés

L'objectif de cette étape est de découper l'antécédent de la règle en sous-ensembles d'atomes liés pouvant être traités comme un tout par un des raisonneurs. Ces ensembles regroupent les atomes du même type partageant certaines variables.

Il est préférable que le raisonneur évalue plusieurs atomes en même temps pour plusieurs raisons.

Premièrement, lorsqu'il traite des atomes représentant des concepts *OWL* liés, le raisonneur est plus performant. Pour évaluer cette composante, le raisonneur consulte la base de connaissances et retourne un ensemble de substitutions possibles pour ces atomes. L'information nécessaire à la construction de ces substitutions est stockée soit dans un fichier, soit dans une base de données. A chaque requête, le raisonneur accède à la structure de données. Les atomes liés ont des variables en commun car ils permettent l'extraction d'informations concernant un individu particulier. Bien souvent ces informations sont stockées en un bloc dans la structure de données. A la place de construire une requête pour chacun des atomes, d'interroger plusieurs fois la structure de données puis d'unifier les substitutions trouvées, le raisonneur construit une seule requête et réduit ainsi le nombre d'accès à cette structure de données.

La seconde raison n'est pas simple. Lorsque le raisonneur traite plusieurs atomes *OWL* liés, il est capable d'effectuer des recoupements logiques. Le raisonneur utilise les relations définies dans l'ontologies afin d'optimiser la découverte de l'information. Par exemple, lorsqu'une composante contient les atomes $Man(X)$ et $Parent(X)$, le *dl_reasoner* optimise cette requête en ne recherchant que les pères dans la base de connaissances.

Pour illustrer la construction des composantes d'atomes liés, la règle suivante est prise en exemple.

Règle 4.2.2 ***Def-hastenyearsmore***

$$\begin{aligned} hasTenYearsMore(?Person1, ?Person2) \leftarrow\ & hasAge(?Person1, ?Age1) \\ & \wedge\ Person(?Person1) \\ & \wedge\ hasAge(?Person2, ?Age2) \\ & \wedge\ Person(?Person2) \\ & \wedge\ swrlb : add(?Age1, ?Age2, 10) \end{aligned}$$

L'analyse de cette règle identifie les 3 composantes suivantes :
- $hasAge(?Person2, ?Age2) \wedge Person(?Person2)$
- $swrlb : add(?Age1, ?Age2, 10)$
- $hasAge(?Person1, ?Age1) \wedge Person(?Person1)$

4.2.3 La construction du graphe des dépendances entre les composantes d'une règle

La construction de ce graphe est utile afin de fixer l'ordre d'évaluation des composantes identifiées lors de l'étape précédente. Elle permet aussi de déterminer les substitutions des variables nécessaires à l'évaluation des autres atomes de la règle.

A l'instar du graphe de dépendances des règles construit pour l'évaluation de l'ensemble des règles, le graphe de dépendances entre les composantes de règles est constitué d'un ensemble de noeuds et d'arcs. Chaque composante est placée dans un noeud. Les arcs fixent les liens entres les noeuds. Un arc relie un noeud d'une composante de *dl_predicate* et un noeud contenant une composante avec des atomes de type *built_in* ou *differentFrom* si les atomes de leur composante ont des variables communes. Les arcs ont toujours un noeud de départ contenant une composante d'atomes de *dl_predicate* et un noeud d'arrivée avec une composante de type *built_in* ou *differentFrom*. Le nom de l'arc est la (les) variable(s) commune(s) aux deux composantes. Cet arc exprime le fait que d'une part les atomes de type *buil_in* et *differentFrom* sont à évaluer en dernier lieu et d'autre part que les substitutions obtenues lors de l'évaluation de la composante du noeud de départ de cet arc portant sur les variables contenues dans le nom de cet arc sont nécessaires à l'évaluation de la composante du noeud d'arrivée. Dans ce graphe, les noeuds contenant des composantes dont les variables des atomes n'ont pas de lien avec d'autres composantes ne font pas partie d'un arc.

La figure 4.3 (voir 63) est le graphe de dépendances construit à partir de la règle 4.2.2. Le nom de chaque arc fait référence aux substitutions de la variable à conserver lors de l'évaluation de la composante du noeud de départ de cet arc.

4.2.4 La détermination de l'ordre d'évaluation

A partir du graphe des dépendances des composantes, il est aisé de déterminer un ordre d'évaluation. La détermination de cet ordre garantit l'efficacité du raisonnement comme expliqué dans l'exemple page 59. Une composante est donc évaluée si les composantes dont elle dépend l'ont déjà été. Il a été choisi de traiter en premier lieu les composantes reprises

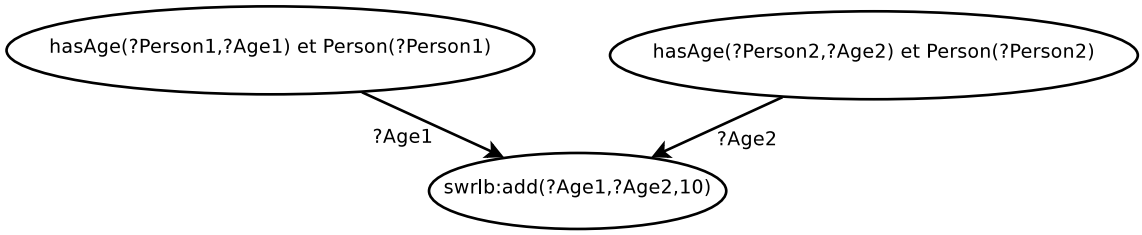

FIG. 4.3 – Le graphe de dépendances entre les composantes de la règle 4.2.2

dans les noeuds isolés car les substitutions obtenues lors de l'évaluation des atomes de ces composantes ne sont pas utiles au traitement d'autres composantes. Vient ensuite le traitement des composantes des noeuds de début des arcs contenant les atomes de type *dl_predicate*. Lors de l'évaluation d'une telle composante, il est utile de garder en mémoire les substitutions dont les variables figurent dans le nom de l'arc joignant ce noeud. Elles vont effectivement servir pour le traitement de la composante du noeud d'arrivée de l'arc. Finalement, ce sont les composantes de type *built_in* et *differentFrom* des noeuds restants qui doivent être évaluées. Cet ordre respecte bien les dépendances entre les composantes ainsi que le fait que les atomes de type *built_in* et *differentFrom* liés à des composantes de *dl_predicate* sont évalués en dernier lieu.

La construction du graphe des dépendances entre les composantes d'une règle et la détermination de l'ordre d'évaluation de ces composantes sont à effectuer une fois pour toutes les évaluations d'une règle. Dès lors, il est intéressant de retenir le résultat de ces deux opérations afin de ne les exécuter qu'une seule fois, avant le traitement de l'ensemble des règles.

L'ordre d'évaluation fixé à partir de la figure 4.3 (voir page 63) détermine l'ordre d'évaluation suivant :

1. $hasAge(?Person1, ?Age1) \ \wedge \ Person(?Person1)$
2. $hasAge(?Person2, ?Age2) \ \wedge \ Person(?Person2)$
3. $swrlb : add(?Age1, ?Age2, 10)$

La permutation des composantes 1 et 2 est possible car elles sont toutes les deux constituées de *dl_predicate*.

4.2.5 L'évaluation des composantes

Chaque composante contient des atomes d'un même type à évaluer via un raisonneur particulier. Pour chacun de ces types, le procédé d'évaluation des atomes diffère et doit donc être détaillé.

L'évaluation des atomes de type *dl_predicate*

A cette étape du processus, les composantes contenant des atomes de type *dl_predicate* sont évaluées. Pour chacune d'elles, une requête (une *abox_query*)[1] est construite afin d'interroger la base des faits *abox* (voir page 31). Le *dl_reasoner* cherche dans cette base de faits *abox* toutes les instanciations possibles des atomes composant la requête. Si le *dl_reasoner* dispose de l'information suffisante pour évaluer chaque atome, le résultat de la requête est un ensemble de substitutions. Chacune de ces substitutions détermine une valeur possible de toutes les variables présentes dans les atomes de la requête. Les substitutions de toutes les variables ne doivent pas être retenues. Seules celles qui portent sur des variables utilisées dans d'autres atomes sont mémorisées. Deux cas sont possibles ;

- Les substitutions portent sur des variables présentes dans une autre composante de type *built_in* ou *differentFrom*. Cette information est fournie par le graphe de dépendances des composantes. En effet, si une composante est le noeud de départ d'un arc de ce graphe, les substitutions des variables contenues dans le nom de cet arc obtenues lors de l'évaluation de cette composante doivent être sauvées.
- Les substitutions des variables apparaissent dans un atome de la tête de la règle doivent également être enregistrées.

Ces substitutions sont ressorties lors de l'évaluation de cet autre composante ou de cet autre atome. Les autres substitutions ne sont plus d'aucune utilité et peuvent donc être mises de côté. Si le *dl_reasoner* ne dispose pas de l'information suffisante pour déterminer les substitutions de chaque atome de la requête, l'évaluation de la règle est stoppée et aucun nouveau fait n'est découvert.

Afin d'illustrer l'évaluation des atomes de type *dl_predicate*, la base de faits *abox* reprise dans le tableau 4.1 (voir page 65) est associée à la règle 4.2.2 rappelée ci-dessous.

$$\begin{aligned} hasTenYearsMore(?Person1, ?Person2) \leftarrow\ & hasAge(?Person1, ?Age1) \\ & \wedge\ Person(?Person1) \\ & \wedge\ hasAge(?Person2, ?Age2) \\ & \wedge\ Person(?Person2) \\ & \wedge\ swrlb : add(?Age1, ?Age2, 10) \end{aligned}$$

L'évaluation des composantes des deux noeuds de départ des deux arcs du graphe 4.3 (voir page 63) fournit les substitutions suivantes :

$\{?Person1/Paul, ?Age1/20;\quad ?Person1/Bob, ?Age1/30;\quad ?Person1/Alice, ?Age1/30\}$
pour la première composante

$\{?Person2/Paul, ?Age2/20;\quad ?Person2/Bob, ?Age2/30;\quad ?Person2/Alice, ?Age2/30\}$
pour la seconde composante

[1] Cette transformation garantit que tous les atomes de la composante sont bien du type *dl_predicate*. Si ce n'est pas le cas, cette transformation échoue.

Faits abox
Person(Paul)
Person(Bob)
Person(Alice)
hasAge(Paul, 20)
hasAge(Bob,30)
hasAge(Alice,30)

TAB. 4.1 – Les connaissances relatives à la règle 4.2.2

Les substitutions des variables $Age1$ et $Age2$ sont retenues car elles vont être utilisées lors de l'évaluation de l'atome *built_ in* de la règle. Dans le graphe de la figure 4.3, ces variables sont reprises dans le nom des arcs les reliant au noeud contenant l'atome *built_ in*.

L'évaluation des atomes de type *built_ in* et *differentFrom*

Les atomes *built_ in* et *DifferentFrom* dans les composantes sont traités un à un via un raisonneur. Le travail de ce raisonneur est quelque peu différent de celui du *dl_ reasoner*. L'évaluation d'un atome ne consiste pas à le comparer à d'autres atomes figurant dans une base de connaissances. Pour chaque substitution, les variables des arguments de l'atome à évaluer doivent être remplacées par leurs valeurs dans la substitution. Ensuite, cet atome est passé au raisonneur qui décide si ces valeurs sont correctes en fonction de la relation *built_ in* ou *differentFrom* considerée.

Le raisonneur développé à l'entreprise *Mission Critical* prend un atome de type *built_ in* ou *differentFrom* en entrée, il l'évalue et peut renvoyer cinq résultats différents pouvant être interprétés comme suit :

Substitutions **:** Ce résultat est renvoyé lorsque le raisonneur doit évaluer un atome de type *built_ in* contenant une inconnue. L'argument *Substitutions* est l'ensemble des substitutions possibles pour cette inconnue ;

Rien : Aucune solution ne peut être trouvée, ni maintenant ni après d'autres évaluations. La substitution des variables de l'atome ne respecte pas la signification du prédicat de l'atome ;

La substitution vide : Cela signifie que tout s'est passé correctement et que le raisonneur a évalué l'atome. La substitution des variables de l'atome respecte le prédicat ;

unbound **:** Le raisonneur est incapable de résoudre ce problème pour l'instant car l'information dont il dispose est insuffisante. Il se peut qu'une solution réside mais d'autres atomes doivent d'abord être évalués. Aprés avoir exécuté d'autres substitutions cet atome doit être repassé au raisonneur ;

Not Supported **:** signifie que le type de l'atome ne figure pas dans les opérations réalisables dans le langage *SWRL*.

Plusieurs comportements sont possibles en fonction du résultat renvoyé par le raisonneur.

S'il renvoie *Not Supported*, l'évaluation de toutes les règles doit être stoppée car le format de l'atome n'est pas reconnu pas le raisonneur.

Le résultat *unbound* est une limitation du raisonneur survenant lors de l'évaluation d'un atome de type *differentFrom* ou d'un atome de type *built_in*. Ce résultat est renvoyé lorsque les arguments d'un atome *differentFrom* n'ont pas tous une valeur dans la substitution. Pour les atomes de type *built_in*, ses arguments doivent tous avoir une valeur dans les substitutions, à l'exception d'un des arguments dont la valeur est déterminée par l'évaluation de cet atome. S'il y a plus d'une variable sans valeur, le raisonneur retourne *unbound*. Dans le cas où il ne reste en entrée que des atomes pour lesquels le raisonneur renvoie *unbound*, aucune solution ne pourra être trouvée et la règle ne peut pas être évaluée. Cette limitation du raisonneur est développée plus en détails dans la section réservée à la critique de l'engin.

La réception de *Rien* signifie que la substitution des variables ne respecte pas la signification de l'atome. Cette substitution doit être mise de côté et les autres substitutions doivent être évaluées. Si après le traitement de ces dernières, le résultat ne change pas, cela signifie qu'aucune substitution ne convient pour cet atome. L'évaluation de cette règle n'a donc pas de solution et peut être interrompue. Par exemple, l'évaluation de l'atome $swrlb : add(35, 22, 10)$ retourne *Rien* car 22 plus 25 est différent de 35.

Si le raisonneur renvoie la *Substitution vide* ou des *Substitutions*, cela signifie que la substitution des variables respecte la définition de l'atome évalué. Lorsque de nouvelles *Substitutions* sont trouvées, elles apportent de l'information supplémentaire. Elles doivent donc être unifiées avec les substitutions trouvées lors de l'évaluation des autres composantes.

Afin d'illustrer ces propos, l'atome $swrlb : add(?Age1, ?Age2, 10)$ contenu dans la composante du graphe 4.3 (voir page 63) est évalué. Avant de pouvoir traiter cet atome, il faut remplacer les occurrences des variables $Age1$ et $Age2$ par leurs valeurs données par les substitutions. Ces valeurs sont extraites de l'unification des substitutions trouvées lors de l'évaluation des composantes contenant des atomes de type *dl_predicate*. Le résultat de l'unification des substitutions découvertes page 64 est repris ci-dessous.

$$\{?Person1/Paul, ?Age1/20, ?Person2/Paul, ?Age2/20;$$
$$?Person1/Paul, ?Age1/20, ?Person2/Bob, ?Age2/30;$$
$$?Person1/Paul, ?Age1/20, ?Person2/Alice, ?Age2/30;$$
$$?Person1/Bob, ?Age1/30, ?Person2/Paul, ?Age2/20;$$
$$?Person1/Bob, ?Age1/30, ?Person2/Bob, ?Age2/30;$$
$$?Person1/Bob, ?Age1/30, ?Person2/Alice, ?Age2/30;$$
$$?Person1/Alice, ?Age1/30, ?Person2/Paul, ?Age2/20;$$
$$?Person1/Alice, ?Age1/30, ?Person2/Bob, ?Age2/30;$$
$$?Person1/Alice, ?Age1/30, ?Person2/Alice, ?Age2/30\}$$

Le remplacement des occurrences des variables $Age1$ et $Age2$ dans l'atome $swrlb : add(?Age1, ?Age2, 10)$ donne le résultat suivant :

1. $swrlb : add(20, 20, 10)$
2. $swrlb : add(20, 30, 10)$
3. $swrlb : add(20, 30, 10)$

4. $swrlb : add(30, 20, 10)$
5. $swrlb : add(30, 30, 10)$
6. $swrlb : add(30, 30, 10)$
7. $swrlb : add(30, 20, 10)$
8. $swrlb : add(30, 30, 10)$
9. $swrlb : add(30, 30, 10)$

Seules les atomes 4 et 7 respectent la définition de l'addition. Dès lors, les seules substitutions conservées sont $\{?Person1/Bob, ?Age1/30, ?Person2/Paul, ?Age2/20\}$ et $\{?Person1/Alice, ?Age1/30, ?Person2/Paul, ?Age2/20\}$.

Après l'évaluation de tous les atomes *built_in* et *differentFrom* des composantes, seules les substitutions concernant des variables apparaissant dans les atomes du conséquent doivent être retenues. Les autres ne servent à rien. Dans l'exemple décrit ci-dessus, seules les substitutions des variables $Person1$ et $Person2$ sont retenues car elles sont utilisées pour évaluer le conséquent de la règle 4.2.2 (voir page 62). Les substitutions des variables $Age1$ et $Age2$ peuvent être écartées.

4.2.6 L'évaluation du conséquent

L'évaluation du conséquent se fait à partir des substitutions des variables obtenues par l'évaluation de l'antécédent. Toutes les variables contenues dans les atomes de la tête sont substituées par leurs valeurs dans chaque substitution. L'interprétation des atomes présents dans le conséquent d'une règle se fait en fonction de leur type. Les prédicats de type *built_in*, *sameAs* et les *différentFrom* sont des contraintes d'intégrité posées sur la règle tandis que les *dl_predicates* sont les faits à ajouter à la base de connaissances. Si pour une variable d'un atome du conséquent il n'existe aucune valeur dans les substitutions, l'évaluation de la règle est interrompue. En effet, la sémantique de *SWRL* impose que le corps de la règle fixe les valeurs des variables. Si une variable n'apparaît pas dans un atome du corps, cela signifie que cette règle est valable pour toutes les valeurs possibles de cette variable. L'implémentation actuelle de *SWRL* à *Mission Critical* ne permet pas de représenter une telle contrainte. Ceci constitue dès lors une limitation de l'implémentation du langage abordée dans une partie ultérieure de ce mémoire.

Les contraintes d'intégrité

Les atomes de type *built_in*, *sameAs* et les *différentFrom* présent dans la tête d'une règle sont des contraintes sur les substitutions des variables obtenues lors de l'évaluation du corps de cette règle. Ces contraintes imposent à certaines variables d'être identiques, différentes, égales à une constante près etc. Si elles ne sont pas respectées, l'évaluation de la règle échoue.

Les atomes de type *sameAs* Si les deux variables n'ont pas les mêmes valeurs dans toutes les substitutions, alors une contrainte d'intégrité est violée.

Les atomes de type *built_in* ou *differentFrom* L'évaluation de tous les atomes doit réussir pour chaque substitution sans quoi la contrainte d'intégrité est violée. Si de

nouvelles substitutions sont trouvées, une erreur doit être renvoyée car toutes les variables doivent être fixées dans le corps.

La règle ci-dessous est correctement évaluée à condition que tous les frères aient le même père. Sinon l'évaluation échoue.

Règle 4.2.3

$$\begin{aligned} sameAs(?Z, ?W) \leftarrow\ & hasBrother(?X, ?Y) \\ & \wedge hasParent(?Y, ?Z) \\ & \wedge hasParent(?X, ?W) \\ & \wedge Father(?W) \\ & \wedge\ Father(?Z) \end{aligned}$$

L'inférence de nouveaux faits

L'évaluation des atomes de type *dl_predicate* du conséquent enrichit la base de connaissances de nouveaux faits. Comme il a été vu précédemment, toutes les variables du prédicat doivent avoir une valeur dans les substitutions. Si c'est le cas, pour chacune de ces substitutions, un nouveau fait est découvert en remplaçant les variables de ce prédicat par leur valeur dans la substitution. Chacun de ces nouveaux faits sont ajoutés à la base de connaissances.

Afin d'illustrer ces propos, la règle 4.2.2 (voir page 62) est prise en exemple. Il a est utile de d'abord rappeler cette règle.

$$\begin{aligned} hasTenYearsMore(?Person1, ?Person2) \leftarrow\ & hasAge(?Person1, ?Age1) \\ & \wedge\ Person(?Person1) \\ & \wedge\ hasAge(?Person2, ?Age2) \\ & \wedge\ Person(?Person2) \\ & \wedge\ swrlb : add(?Age1, ?Age2, 10) \end{aligned}$$

Les substitutions des variables obtenues lors de l'évaluation de l'antécédent de cette règle sont $\{?Person1/Bob, ?Age1/30, ?Person2/Paul, ?Age2/20\}$ et $\{?Person1/Alice, ?Age1/30, ?Person2/Paul, ?Age2/20\}$.

L'évaluation du conséquent de cette règle découvre les faits $hasTenYearsMore(Bob, Paul)$ et $hasTenYearsMore(Alice, Paul)$. Ces nouveaux faits sont ajoutés aux faits *abox* de la base de connaissances.

4.2.7 La mise à jour de la base des faits

Le résultat de l'évaluation du conséquent permet de mettre à jour la base des faits. Les atomes obtenus sont transformés en *triples rdf* avant d'être ajoutés à la base de connaissances *abox*.

4.3 Les critiques de la méthodologie

La méthodologie présentée précédemment détaille les étapes à suivre pour évaluer un ensemble de règles *SWRL* dans le cadre d'une ontologie *OWL*. Dans cette partie du travail, les atouts, les inconvénients et les limitations de cette méthodologie sont abordés.

4.3.1 Les atouts de la méthode

L'étude des dépendances entre les règles

La construction du graphe des dépendances entre les règles et la détermination d'un ordre partiel garantissent qu'une règle n'est évaluée que lorsque toutes les règles pouvant générer de nouvelles connaissances utiles à l'évaluation de l'antécédent de cette règle ont déjà été évaluées (Ce raisonnement ne s'applique pas à l'évaluation des règles définissant un circuit). Sans l'étude du graphe de dépendances, les règles seraient traitées selon un ordre quelconque et des faits faisant référence à des concepts *OWL* utiles à l'évaluation de certaines règles seraient découverts trop tard pour être pris en compte. Ce graphe détermine un ordre de traitement de ces règles en respectant les dépendances entre celles-ci afin que l'information générée par leur évaluation soit maximale.

L'évaluation d'une règle

En suivant la méthode proposée, les atomes liés sont regroupés afin d'être évalués ensemble par les différents raisonneurs. Cette opération réduit le nombre d'accès aux structures de données et tire pleinement partie des recoupements logiques dont sont capables les raisonneurs sous-jacents.

La construction du graphe de dépendances entre les composantes d'une règle permet de déterminer un ordre d'évaluation des composantes et d'identifier les substitutions à mémoriser utilisées lors de l'évaluation d'autres atomes.

L'opération de sélection des substitutions nécessaires à l'évaluation d'autres atomes est en fait une projection sur les variables d'intérêt. Cette optimisation a pour but de minimiser le travail d'unification des substitutions issues des différents types de composantes évaluées. La méthode détermine les substitutions respectant l'ensemble des atomes de l'antécédent d'une règle qui sont utiles à l'évaluation du conséquent.

4.3.2 Les inconvénients et les limitations de la méthode

L'étude des dépendances entre les règles

Comme il a été vu dans le chapître 3, l'évaluation d'un ensemble de règles selon l'algorithme 1 en chaînage avant (voir page 36) a pour objectif de vérifier la satisfaction d'un but. Ce but est un atome ou bien l'instanciation partielle d'un atome à vérifier. L'algorithme d'évaluation des règles s'arrête lorsque cet objectif est atteint.

La méthodologie proposée dans ce mémoire effectue toutes les inférences possibles même si elles n'aident pas à atteindre le but fixé. La méthode n'est pas orientée vers la vérification d'un but car cela n'avait pas été demandé lors de son élaboration. L'objectif du stage réalisé à *Mission Critical* était d'implémenter un raisonneur capable d'évaluer un ensemble de règles *SWRL* en chaînage avant et d'inférer tous les faits possibles. Il faut préciser que cette particularité du raisonneur ne garantit pas la terminaison de l'évaluation de l'ensemble des règles. Tant que le point fixe du processus d'inférence n'est pas atteint, le raisonneur continue l'évaluation des règles.

La méthode décrite dans ce mémoire peut être modifiée afin qu'elle s'arrête lorsqu'un but particulier est vérifié. Pour l'adapter, il ne suffit pas de comparer les motifs des nouveaux faits inférés lors de l'évaluation des règles avec le but à satisfaire car ces faits font partie d'une ontologie. Pour vérifier la satisfaction d'un but, il faut utiliser le raisonneur de la logique de description (à savoir le *dl_ reasoner*) car celui-ci fait des recoupements logiques entre les différentes classes de l'ontologie. Par exemple, pour vérifier le but $Father(bob)$, il ne faut pas nécessairement que cet atome soit découvert par le moteur d'inférence. Il suffit que lors de l'évaluation des règles, les faits $Parent(bob)$ et $Man(bob)$ soit inférés. Le *dl_ reasoner* utilise la relation entre les classes *Parent* et *Man* de l'ontologie afin de déduire que *bob* est bien père.

L'évaluation d'une règle

L'ordre d'évaluation des composantes calculé permet de minimiser le nombre de substitutions d'une variable comme il a été expliqué dans l'exemple page 59. Le raisonneur implémenté à *Mission Critical* ne sait évaluer les atomes de type *built_ in* et de *differentFrom* que sous certaines conditions relatives à l'instanciation des arguments. Cette limitation oblige à instancier au maximum les atomes de type *built_ in* et *differentFrom* avant de faire appel au raisonneur.

4.3.3 La comparaison avec les autres méthodes

Les deux principales méthodes d'évaluation de règles en chaînage avant dominant la littérature sont la transformation *en ensembles magiques* (voir section 3.3 page 41) et l'algorithme *Rete* (voir section 3.4 page 44). Après les avoir analysées, il est intéressant de se demander si elles peuvent être intégrées à la méthodologie proposée dans ce mémoire.

La transformation en *ensembles magiques*

La transformation en *ensembles magiques* semble pouvoir être intégrée à la méthode d'évaluation en chaînage avant présentée au début de ce chapître. En effet, les auteurs de l'article [MVM03] expliquent qu'ils voient la transformation en *ensembles magiques* comme *une technique prometteuse optimisant la recherche de la réponse à une requête* concernant les concepts d'une ontologie.

L'intégration de ce processus à la méthode proposée pourrait se faire de manière aisée. Avant de construire le graphe de dépendances entre les règles, il suffirait de générer

les nouvelles règles en suivant la transformation en *ensembles magiques* comme expliquée dans la section 3.3.2 (voir page 42) de ce mémoire. Lorsque les règles seraient transformées, la méthode proposée dans le début de ce chapître serait appliquée au nouvel ensemble de règles. Les conséquence de cette transformation sont expliquées dans la partie 3.3 (voir page 41).

Ces propos peuvent être illustrés à partir des règles énoncées dans la section 4.1 (voir page 52). Il est supposé que l'objectif de l'évaluation de ces règles est de trouver tous les ancêtres de *john* ($hasAncestor(john, Y)$). La transformation en *ensembles magiques* de ces règles concerne uniquement les règles 4.1.6 et 4.1.5 car elles sont les seules à participer à la vérification du but. Leur transformation est décrite dans la section 3.3.2. Les autres règles restent telles quelles. Le graphe des dépendances entre les nouvelles règles et les règles inchangées est présenté dans la figure 4.4 (voir page 71).

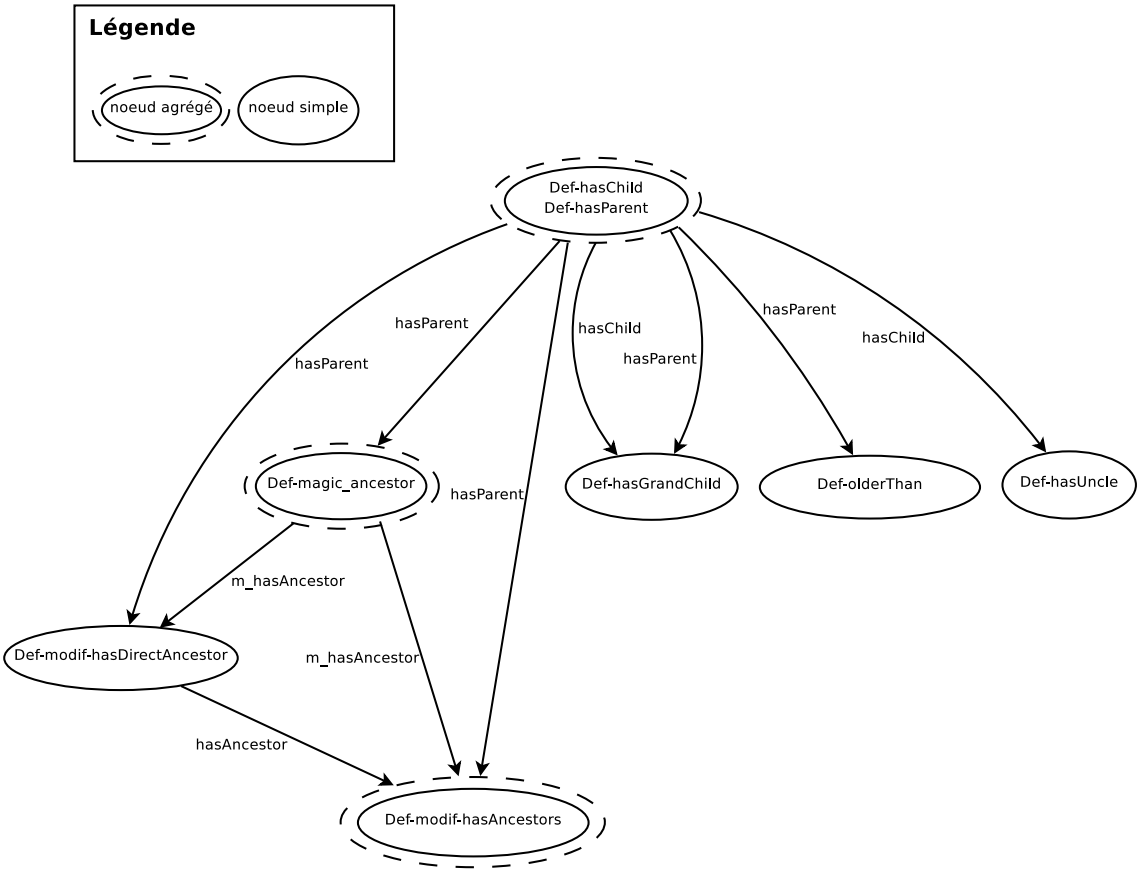

FIG. 4.4 – Le graphe des noeuds agrégés des dépendances entre les règles transformées en *ensembles magiques*

Dans la figure 4.4, la correspondance entre les noeuds du graphe et les nouvelles règles énoncées page 43 est la suivante :

- $Def - magic - ancestor$ correspond à la règle (3.1) ;
- $Def - modif - hasDirectAncestor$ correspond à la règle (3.2) ;
- $Def - modif - hasAncestors$ correspond à la règle (3.3).

Les autres noeuds du graphe contiennent les noms des règles n'ayant pas été modifiées. L'évaluation des règles présentes dans les noeuds se fait comme expliqué précédemment dans la méthode d'évaluation d'un ensemble de règles (voir page 52).

Toutefois, il serait intéressant d'analyser plus en détails la possibilité d'intégrer la transformation en *ensembles magiques* à la méthode d'évaluation des règles *SWRL* afin de s'assurer qu'elle conserve bien les spécificités propres aux raisonnements sur les ontologies.

L'algorithme *Rete*

Comme il a été expliqué auparavant, l'algorithme *Rete* construit un réseau de noeuds à partir des règles et des faits connus. Cette structure sert de base à l'évaluation de ses composants effectuée en comparant les motifs. Cette méthode paraît difficile et moins performante à appliquer au langage de règles *SWRL* car celui-ci fait référence à des concepts définis dans des ontologies. Lors de l'évaluation d'une règle, la méthode proposée précédemment analyse les liens entre les atomes présents dans cette règle. Lorsque plusieurs atomes de type *dl_predicate* sont liés, ils sont passés ensemble au *dl_reasoner* car celui-ci fait des recoupements logiques entre les concepts *OWL*. L'algorithme *Rete* quant à lui ne tient pas compte des liens existant entre les atomes car il les évalue en comparant les motifs. Par conséquent, lors de l'évaluation d'une règle, l'information découverte par cet algorithme peut ne pas être maximale. L'algorithme *Rete* ne semble donc pas adapté à l'évaluation d'un ensemble de règles *SWRL*.

Chapitre 5

Le conclusion et les perspectives

Ce travail a permis d'exposer une méthode capable d'évaluer un ensemble de règles *SWRL* et de découvrir toutes les nouvelles connaissances inférables à partir de ces règles. Cette méthode a été implémentée dans le langage déclaratif *Mercury* lors du stage à l'entreprise *Mission Critical*. Il a été vu dans le partie 4.3 (voir page 69) que certaines améliorations pouvaient être intégrées à cette méthode afin d'optimiser le traitement des règles.

Un point important de la méthode d'évaluation en chaînage avant d'un ensemble de règles n'a pas été abordé dans ce mémoire. Il s'agit des performances. Celles-ci n'ont pas été testées pendant le stage à l'entreprise *Mission Critical*. Toutefois, il serait judicieux de comparer le temps de traitement nécessaire à cette méthode pour évaluer un ensemble de règles avec le temps mis par d'autres moteurs d'inférences implémentés. Les plus connus sont *RACER* [Rac08] et *Pellet* [Pel08].

L'analyse de l'algorithme *Rete* a mis en évidence un point important : les algorithmes de chaînage avant évaluant les règles en comparant les motifs ne sont pas adaptés à *SWRL*. Les spécificités de ce langage, et plus particulièrement du langage *OWL*, nécessitent un traitement plus élaboré. Plus précisément, les raisonneurs évaluant les atomes d'une règles en comparant les motifs sont incapables de faire les recoupements logiques entre les classes définies par les relations *OWL* afin de trouver tous les faits correspondant à ces atomes.

Il a aussi été expliqué que la transformation en *ensembles magiques* semble pouvoir améliorer la méthode détaillée précédemment. Cette technique transformerait les règles *SWRL* afin d'intégrer le but à satisfaire. Dès lors, de nombreux faits inutiles ne seraient pas inférés. Dans la littérature informatique, il existe d'autres méthodes permettant d'optimiser l'évaluation d'un ensemble de règles. La méthode d'*évaluation partielle* [JGS99] est adaptée au traitement des langages de la logique du premier ordre. Cette méthode analyse les règles d'un programme statiquement dans le but de les transformer en un ensemble de règles plus efficaces. Dans la méthode décrite dans ce mémoire, il a été explicité que lorsque le *dl_reasoner* évalue un ensemble d'atomes liés de type *dl_predicate*, il optimise la requête qu'il adresse à la base des connaissances en tenant compte des relations entres les classes *OWL*. Lors de l'analyse des règles, la méthode d'*évaluation partielle* est capable d'identifier ces relations et de transformer les règles statiquement. Il serait intéressant de se pencher

sur cette technique d'optimisation en vue de l'intégrer dans la méthode proposée dans ce mémoire.

Bibliographie

[Alb89] L. Albert, *Complexité en moyenne de l'algorithme de multi-pattern matching rete sur des ensembles de patterns et d'objets de profondeur un*, Rapport de recherche numéro 1009, Unite de recherche INRIA-Rocquencourt, 1989.

[AS06] H. P. Alesso and C. F. Smith, *Thinking on the web, berners-lee, gödel, and turing*, Wiley-Interscience, 2006.

[ASM80] J. Abrial, S. A. Schuman, and B. Meyer, *A specification language, in on the construction of programs*, second edition ed., Cambridge University Press, eds. A. M. Macnaghten and R. M. McKeag, 1980.

[BCM+03] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Sheider, *The description logic handbook : Theory, implementation and applications*, Cambridge University Press, 2003.

[Bir98] R. Bird, *Introduction to functional programming using haskell*, 2nd edition ed., Prentice Hall, 1998.

[Bos01] M. Vanden Bossche, *High-quality and predictble, mission-critical software*, 2001.

[BR91] C. Beeri and R. Ramakrishnan, *On the power of magic*, Journal of Logic Programming **10** (1991), 255–299.

[BV04] D. Brickley and R. V.Guha, *RDF vocabulary description language 1.0 : Rdf schema*, `http://www.w3.org/TR/rdf-schema/#rdfnotcite`, 02 2004.

[Car06] J. Cardoso, *The synntactic and the semantic web*, `http://dme.uma.pt/jcardoso/Books/IDEA-SWTTA/Chap01-Sample.pdf`, 2006.

[CMI08] Dublin Core Metadata Initiative, *About the initiative*, `http://dublincore.org`, 01 2008.

[Doo95] R. B. Doorenbos, *Production matching for large learning systems*, Ph.D. thesis, Carnegie Mellon University, 1995.

[fISM08] Publishing Requirements for Industry Standard Metadata, *About PRISM*, `http://www.prismstandard.org`, 03 2008.

[For82] C. Forgy, *Rete : A fast algorithm for the many pattern/many object pattern match problem*, Artificial Intelligence **19** (1982), 17–37.

[GGP03] U. Glässer, R. Gotzhein, and A. Prinz, *Computer networks : The international journal of computer and telecommuications networking*, vol. Volume 42, Elsevier North-Holland, Inc., 2003.

[Gru93] T. Gruber, *A translation approach to portable ontology specifications*, Knowledge Acquisition **5** (1993), 199–220.

[Gru08] Tom Gruber, *Ontology*, `http://tomgruber.org/writing/ontology-definition-2007.htm`, 01 2008.

[HCS+08] F. Henderson, T. Conway, Z. Somogyi, D Jeffery, P. Schachte, S. Taylor, C. Speirs, T. Dowd, R. Becket, and M. Brown, *Mercury programming language, version 0.13.1*, `www.cs.mu.oz.au/mercury/`, 2008.

[Her07] I. Herman, *W3C semantic web activity*, `http://www.w3.org/2001/sw/`, 10 2007.

[HPSB+04] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, *SWRL : A semantic web rule language combining owl and ruleml*, `http://www.w3.org/Submission/SWRL/`, 05 2004.

[Jac08] I. Jacobs, *History*, `http://www.w3.org/Consortium/history`, 01 2008.

[JGS99] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*, Prentice Hall International 1993, 1999.

[Jon90] C. B. Jones, *Systematic software development using VDM*, second edition ed., Prentice Hall International, 1990.

[KM01] M.-R. Koivunen and E. Miller, *Semantic web kick-off seminar*, W3C Semantic Web Activity, 11 2001.

[Llo87] K. W. Lloyd, *Foundations of logic programming*, second, extended edition ed., Springer series in symbolic computation, Springer-Verlag, 1987.

[LÖ08] L. Liu and M. Tamer Özsu, *Encyclopedia of database systems*, Springer-Verlag, 2008.

[MM04] F. Manola and E. Miller, *RDF primer*, `http://www.w3.org/TR/REC-rdf-syntax/`, 02 2004.

[MTH90] R. Milner, M. Tofte, and R. Harper, *The definition of standard ml*, MIT Press, 1990.

[MVM03] B. Motik, R. Volz, and A. Maedche, *Optimizing query answering in description logics using disjunctive deductive databases*, FZI Research Center for Information Technologies, 2003.

[Pel08] Pellet, *Pellet, the open source OWL DL Reasoner*, `http://pellet.owldl.com/`, 2008.

[Rac08] Racer, *Racer Systems GmbH and Co. KG : Products*, `http://www.racer-systems.com/products/racerpro/index.phtml`, 2008.

[RN03] S. Russell and P. Norvig, *Artificial intelligence : A modern approach, 2nd edition*, Pearson Education International, 2003.

[Rul08] RuleMl, *The rule markup initiative*, `http://www.ruleml.org/`, 04 2008.

[Sch01] S. Schneider, *The B-method : an introduction*, Palgrave, Cornerstones of Computing series, 2001.

[SHBL06] N. Shadbolt, W. Hall, and T. Berners-Lee, *The semantic web revisited*, IEEE Computer Society **21** (2006), 96–101.

[Ull89] J. D. Ullman, *Principles of database and knowledge-base systems, volume ii : The new technologies*, Computer science press, 1989.

[vHM04] F. van Harmelen and D. L. McGuinness, *OWL web ontology language : Overview*, `http://www.w3.org/TR/owl-features/`, 02 2004.

[Wal07] C. Walton, *Agency and the semantic web*, Oxford University Press, 2007.